Erweiterung des AQuA-Systems: ODL-Sprachkonstrukte und interaktive Benutzerschnittstelle

David Trachtenherz

Technische Universität München
Fakultät für Informatik

Diplomarbeit

Erweiterung des AQuA-Systems: ODL-Sprachkonstrukte und interaktive Benutzerschnittstelle

Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Dr. Bernhard Schätz
Bearbeiter:	David Trachtenherz
Abgabedatum:	14.11.2003

Selbständigkeitserklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14.11.2003

____

Zusammenfassung

Die vorliegende Diplomarbeit befasst sich im Rahmen des AutoFocus/Quest-Application-Frameworks mit der Erweiterung der *Operation Definition Language (ODL)* um neue Sprachkonstrukte und der Entwicklung einer interaktive Benutzerschnittstelle für die Durchführung von Benutzereingaben bei der Auswertung von ODL-Abfragen. Nach der Vorstellung des Projektumfeldes und der konzeptionellen Grundlagen von ODL werden die realisierten Erweiterungen beschrieben und eine ausführliche Beschreibung der technischen Implementierung gegeben. Ferner wird eine Reihe von Vorschlägen zur Weiterentwicklung des ODL-Systems gemacht. Abschließend werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf zukünftige Entwicklungsmöglichkeiten gegeben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Das AutoFocus/Quest Application (AQuA) Framework stellt mit der Operation Definition Language (ODL) eine Sprache zur Transformation von AutoFocus/Quest-Modellen zur Verfügung. Die Sprache stellt eine Erweiterung der Prädikatenlogik erster Stufe dar, definiert auf dem Metamodell von AutoFocus/Quest. Ähnlich zu OCL erlaubt sie die Definition von Konsistenzbedingungen. Zusäztlich eignet sie sich dank ihres operationellen Charakters zur Definition von Operationen auf AutoFocus/Quest-Modellen. Um die bei der Ausführung von Operationen notwendigen Benutzerinteraktionen (z.B. Auswahl von Komponenten, Eingabe von Strings) durchführen zu können, soll der ODL-Interpreter eine Benutzerschnittstelle automatisch zur Verfügung stellen.

Konkrete Aufgabenstellung:

Im Rahmen der Diplomarbeit sollte das AQuA-System hinsichtlich folgender Aspekte erweitert werden:

- Die Ausdrucksmächtigkeit von ODL sollte erweitert werden.
- Eine interaktive Benutzerschnittstelle für die Auswertung von ODL-Abfragen sollte entwickelt werden.

Für die Erweiterung der Ausdrucksmächtigkeit sollte ODL um folgende Sprachkonstrukte ergänzt werden:

- Produkttypen: Bildung des kartesischen Produkts über bereits definierte Typen.
- Mengenkomprehension: Bildung von Teilmengen bereits definierter Typen, die über eine Restriktionsbedingung charakterisiert werden.
- Mengentypen: Bildung von Kollektionen von Werten bereits definierter Typen.
- Benamte ODL-Prädikate: Definition von parametrierten Prädikaten mit ODL-Termen als Rümpfen.
- Selektorausdrücke: Zugriff auf Relationen von Entitäten und auf Elemente von Produkttypen mittels der Selektoren.

Hierfür waren neben der Erweiterung der ODL-Grammatik die neuen Typen in das ODL-Typsystem zu integrieren und die Evaluationsfunktionen für die neuen und erweiterten Sprachkonstrukte zu realisieren.

Für die Durchführung von Benutzerinteraktionen, die im Laufe der Auswertung von ODL-Abfragen notwendig sein können, war eine Benutzerschnittstelle zu realisieren, welche für jeden ODL-Datentyp die Eingabe von Werten durch den Benutzer ermöglicht. Als optionale Ausbaustufe sollte für die Benutzerführung die Unterstützung der Rückkehr zu früheren Schritten sowie die Möglichkeit des Abbruch einer Operation implementiert werden.

1.2 Gliederung

In diesem Abschnitt wollen wir die Gliederung der vorliegenden Arbeit vorstellen.

Im Kapitel 2 wird ein kurzer Überblick über das AutoFocus/QUEST-Framework gegeben, in dessen Rahmen die Arbeit durchgeführt wurde, und die Motivation für die Entwicklung von ODL erläutert.

Im Kapitel 3 werden die konzeptionellen Grundlagen von ODL sowie die erste ODL-Implementierung beschrieben.

Das Kapitel 4 schildert die durchgeführten Erweiterungen am Sprachumfang und bietet im Abschnitt 4.2 eine Beschreibung der entwickelten interaktiven Benutzerschnittstelle, die auch als Benutzerhandbuch für die Benutzerschnittstelle dient.

Im Kapitel 5 wird die Implementierung der ODL-Spracherweiterung und der interaktiven Benutzerschnittstelle ausführlich beschrieben. Zusätzlich werden im Abschnitt 5.3 die Schritte kurz erläutert, die zur Implementierung weiterer Änderungen durchzuführen sind. Der Abschnitt 5.4 gibt unter Berücksichtigung der Implementierungsaspekte des ODL-Auswertungssystems Hinweise zum Entwurf effizienter ODL-Abfragen.

Das Kapitel 6 befasst sich mit den Verbesserungs- und Weiterentwicklungsmöglichkeiten für das ODL-Auswertungssystem und für die interaktive Benutzerschnittstelle.

Im letzten Kapitel wird eine kurze Zusammenfassung der Ergebnisse der Arbeit und ein Ausblick auf zukünftige Entwicklungsmöglichkeiten gegeben.

1.3 Notation

In der vorliegenden Arbeit werden verschiedene Textformatierungen und Diagramme verwendet, um das Lesen zu erleichern.

Zunächst die Textformate:

- ODL-Abfragen und Java-Quellcode werden in `Maschinenschrift` gesetzt.
- Wörter, die vom laufenden Text abgehoben werden sollen, sind *kursiv* gesetzt.
- Überschriften der Einzelpunkte von Aufzählungen werden meistens **fett** gesetzt.

Klassendiagramme und Sequenzdiagramme verwenden die UML-Notation, wie sie von TogetherJ in der Version 3.83 unterstützt wird.

Kapitel 2

Überblick

In diesem Kapitel werden wir die CASE-Tools AutoFocus und QUEST kurz vorstellen, die das AQuA-Framework bilden, in dessen Rahmen die aktuelle Diplomarbeit durchgeführt wurde. Der Schwerpunkt wird dabei auf dem Modellierungs- und Validierungstool QUEST sowie dem gemeinsamen Metamodell von AutoFocus und QUEST liegen, da das ODL-Modul Bestandteil des QUEST-Tools ist.

Die Abschnitte 2.1 und 2.2 befassen sich mit den Tools AufoFocus und QUEST sowie mit dem für die Modellierung verwendeten Metamodell. Der Abschnitt 2.3 behandelt die Motivation für komplexe Transformationen an Produktmodellen, und damit die Motivation für die Entwicklung von ODL.

2.1 Modellierungs- und Validierungsframework AutoFocus/QUEST

Die Tools AutoFocus und QUEST dienen zur Erstellung und Validierung von Produktmodellen. AutoFocus ist ein graphischer Editor, in dem Modelle erstellt und bearbeitet werden können (Abbildung 2.1). Zusätzlich können Korrektheitsprüfungen für erstellte Modelle durchgeführt werden. Weitergehende Informationen zur Benutzung von AutoFocus finden sich in [Validator] und auf der Homepage des AutoFocus-Projekts [AFHome].

Für die Entwicklung von Modellen unterstützen AutoFocus und Quest folgende Diagrammarten:

- **SSD (System Structure Diagram)**
  Auf diesen Diagrammen wird die Struktur des modellierten Systems dargestellt. Ein System besteht hierbei aus Komponenten und den Kommunikationskanälen zwischen den Komponenten (s. Diagramm auf Abb. 2.1). Jeder Komponente kann dabei eine interne Struktur durch ein SSD zugeordnet werden – damit sind hierarchische Systembeschreibungen möglich.

- **STD (State Transition Diagram)**
  Mithilfe von Zustandsübergangsdiagrammen wird das Verhalten von Systemen und Komponenten beschrieben – über Kommunikationskanäle erhalten Komponenten Eingaben aus der Umgebung und senden Ausgaben an die Umgebung, die entsprechend dem festgelegten Verhalten und abhängig vom aktuellen Zustand der Komponente erzeugt werden. Bei den STD's handelt es sich um erweiterte endliche Automaten, die lokale Variablen der Komponente nutzen dürfen, dessen Verhalten sie beschreiben.

- **DTD (Datatype Definitions)**
  Die vom einem System verwendeten Datentypen werden in einer textuellen Notation definiert. Der Benutzer kann, zusätzlich zu den vordefinierten Basistypen, eigene Datentypen definieren. Datentypen werden zur Deklaration lokaler Variablen in Komponenten sowie zur Definition von Datentypen der übertragenen Daten bei Kommunikationskanälen und Kommunikationsports verwendet.

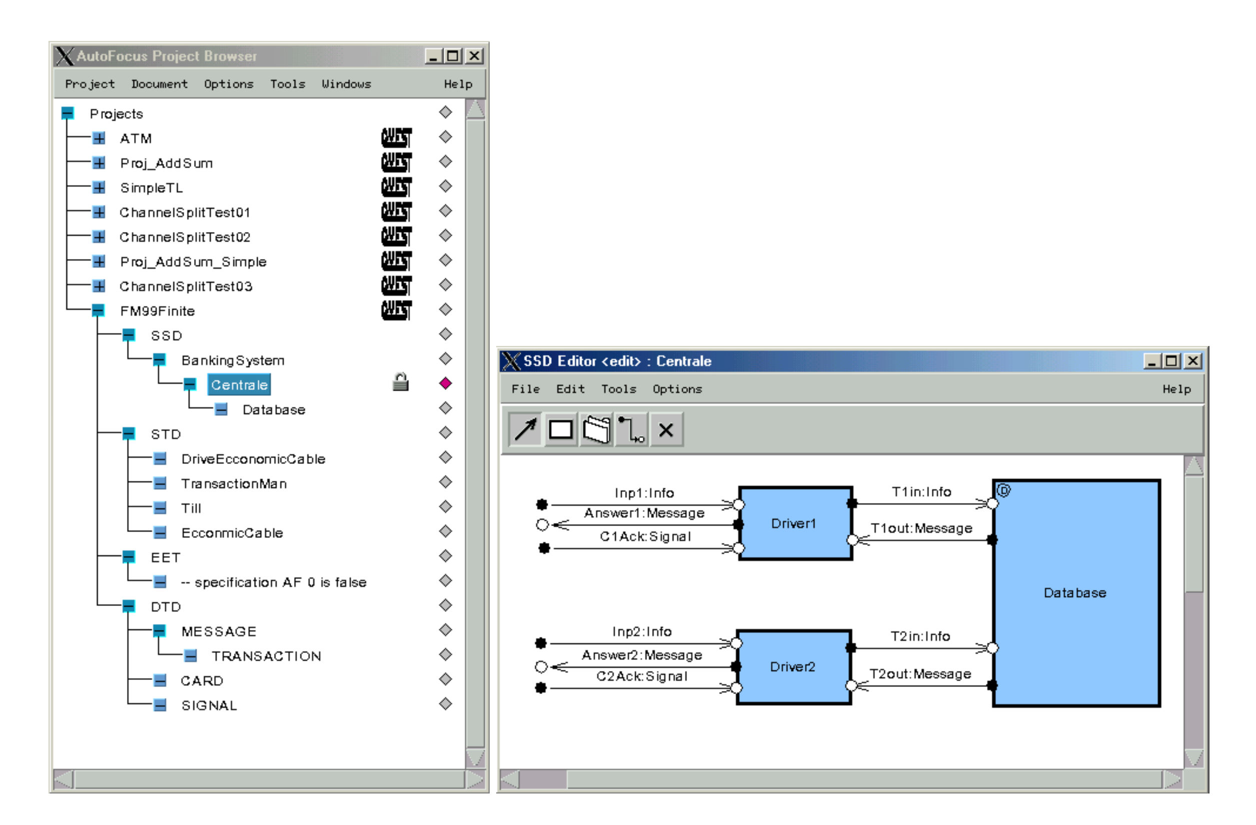


Abbildung 2.1: AutoFocus: Projektbrowser und Diagrammfenster

- **EET (Extended Event Traces)**
  Neben den STD's können auch EET's zur Verhaltensbeschreibung von Komponenten benutzt werden. Sie stellen die kommunikationsorientierte Sicht auf das Verhalten von Komponenten dar, indem sie dieses durch exemplarische Kommunikationsabläufe zwischen Komponenten darstellen. EET's ähneln damit vom Konzept her den Sequenzdiagrammen in UML.

Eine Beschreibung der Architektur und Features von AutoFocus gibt es in [HS02].

QUEST stellt eine Erweiterung von AutoFocus dar, welche die Validierung von Modellen mithilfe existierender formaler Methoden und Werkzeuge ermöglicht, um die Korrektheit kritischer Systemabschnitte zu sichern (s. auch [BLS00]). Das Tool bietet keinen graphischen Editor für Modelle, sondern eine Baumansicht für die Modellstruktur, die aber immer noch Bearbeitungsmöglichkeiten zur Verfügung stellt.

Die Aufgabe von QUEST ist das Testen und die Validierung von Modellen durch den Einsatz von einbindbaren Programmmodulen (Abbildung 2.2), die über die Metamodell-Schnittstelle auf Produktmodelle zugreifen. Zusätzlich ermöglicht QUEST die Generierung von Quellcode verschiedener Programmiersprachen wie C oder Java aus den Produktmodellen.

AutoFocus und QUEST verwenden dasselbe Metamodell und können Produktmodelle untereinander austauschen. Die Abbildung 2.3 zeigt die Komponente, deren Strukturdiagramm auf der Abbildung 2.1 dargestellt ist, als Baumdiagramm im QUEST-Browser.

Ausführliche Informationen zum Arbeiten mit dem QUEST-Tool gibt es im Benutzerhandbuch [QuestUser]. Technische Information zur Implementierung von QUEST liefert das Entwicklerhandbuch [QuestDev].

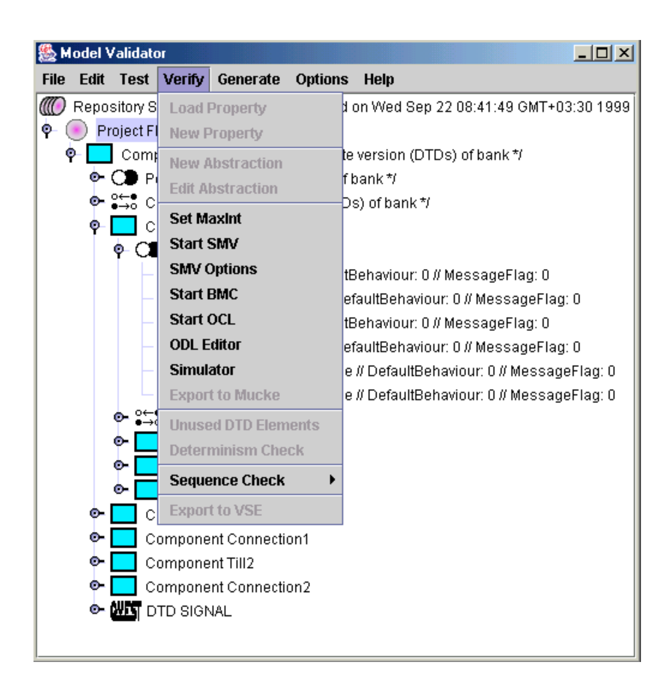


Abbildung 2.2: QUEST: Browser und Verifikationsmenü

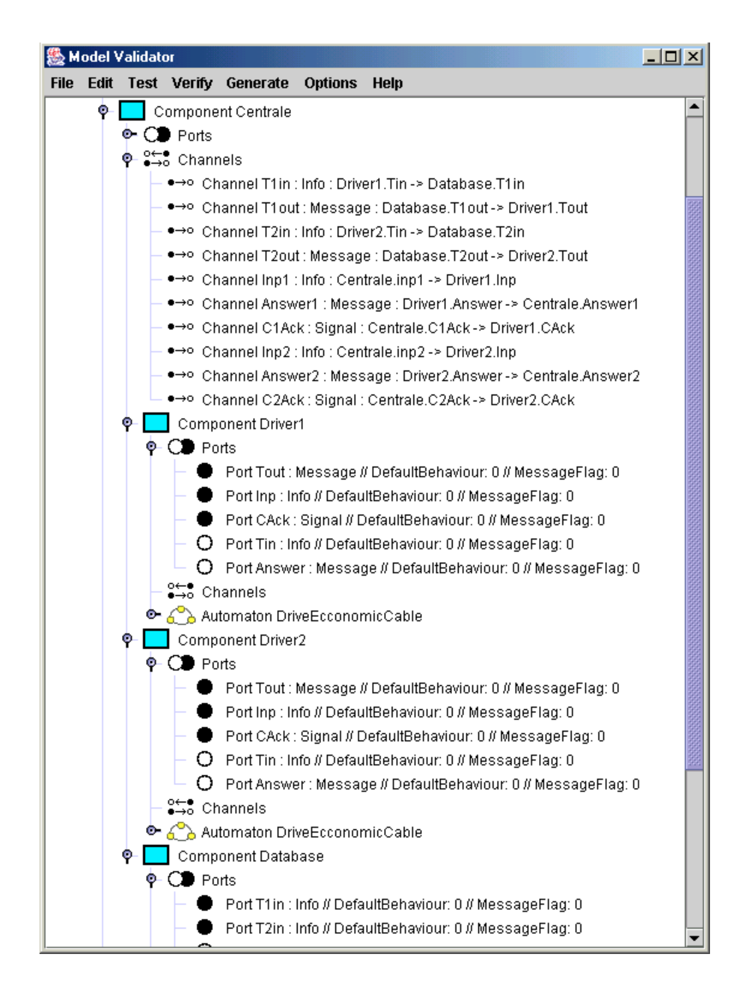


Abbildung 2.3: Baumdarstellung einer Komponente

2.2 Metamodell von AutoFocus/QUEST

Das Metamodell beschreibt, welche Modellierungskonzepte zur Entwicklung von Modellen zur Verfügung stehen (z.B. Komponenten, Kanäle, Ports, Zustände) und wie sie zueinander in Verbindung stehen (z.B. ein Port gehört zu genau einer Komponente) ([Sch01], S.3-4). Die Abbildung 2.4, die aus [TACAS00] entnommen wurde, zeigt ein vereinfachtes Klassendiagramm des Metamodells für SSD's und STD's.
Im Folgenden erläutern wir die für uns wichtigen Elemente des QUEST-Metamodells, die wir in späteren Beispielen von ODL-Abfragen benutzen werden:

- **Komponente**
  Komponenten sind Grundbausteine von Modellen. Eine Komponente enthält Unterkomponenten (das sind Komponenten im SSD, welches die Struktur der betrachteten Komponente beschreibt), Ports, über die Signale gesendet und empfangen werden, sowie Kanäle, die die Unterkomponenten der Komponente untereinander verbinden.

- **Port**
  Ein Port dient einer Komponente zur Kommunikation mit der Umgebung. Das Attribut `Direction` bestimmt, ob der Port ein Ausgangsport oder ein Eingangsport ist. Der Typ der Daten, die der Port senden bzw. empfangen kann, wird bei jedem Port über das Attribut `Type` festgelegt. Ein Eingangsport kann mit höchstens einem Kanal verbunden sein; ein Ausgangsport kann mehrere ausgehende Kanäle haben.

- **Kanal**
  Kanäle verbinden Ports miteinander und ermöglichen damit für Komponenten, zu denen die verbundenen Ports gehören, die Kommunikation untereinander. Ein Kanal verbindet zwei Ports, von denen einer ein Ausgangsport und der andere ein Eingangsport sein muss. Wie schon bei Ports, ist bei jedem Kanal der Typ der übertragenen Daten festgelegt – dieser muss mit dem Datentyp der verbundenen Ports übereinstimmen.

- **Automat**
  Ein Automat beschreibt das Verhalten einer Komponente. Der Automat wird in einem STD erstellt und einer Komponente zugeordnet.

- **Zustand**
  Zustände sind Bestandteile eines Automaten und beschreiben seine Struktur und sein Verhalten. Wie Komponenten, können Zustände hierarchisch aufgebaut sein, d.h., jeder Zustand kann beliebig viele Unterzustände haben, die sein Verhalten bestimmen.

Alle oben aufgeführten Metamodellelemente besitzen das Attribut `Name`. Dies ermöglicht es, Modellelemente unterschiedlich zu benennen, wobei der Name nicht zwingend eindeutig sein muss.

Weiterführenden Informationen zum QUEST-Metamodell gibt es in [BLS01], [QuestDev] und in [TACAS00].

Das Vorhandensein eines gemeinsamen Metamodells ermöglicht es AutoFocus und QUEST, Modelle untereinander ohne Konvertierungsverluste auszutauschen. Des Weiteren können neue Module in QUEST integriert werden, die die Metamodell-Schnittstelle für den Zugriff auf Produktmodellen benutzen – diese Module benötigen keine weiteren Kenntnisse über das QUEST-Tool und sind damit auch in zukünftigen Weiterentwicklungen des AQuA-Frameworks einsetzbar, die das gleiche oder ein abwärtskompatibles Metamodell verwenden.

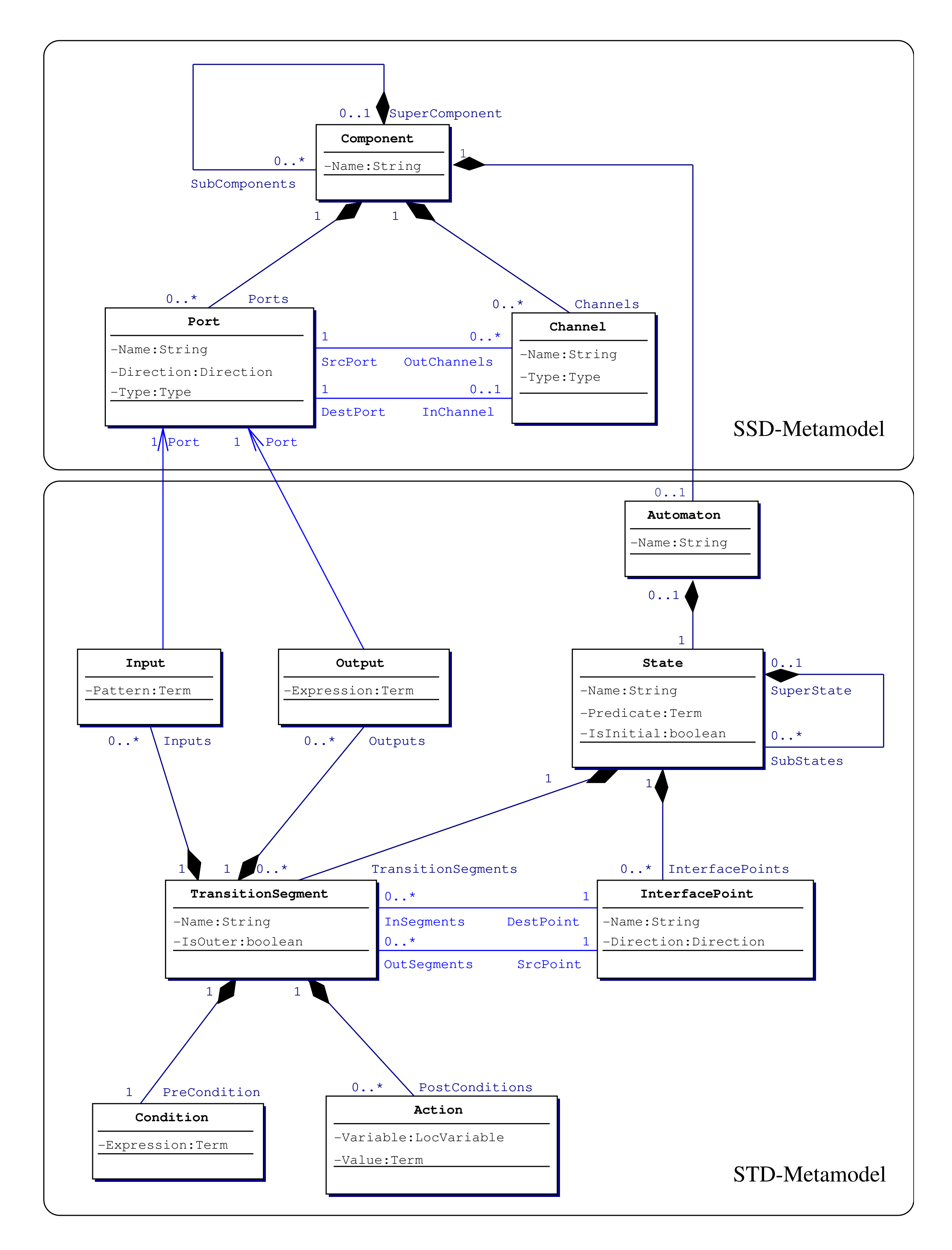


Abbildung 2.4: Integriertes Metamodell für Strukturdiagramme und Zustandsübergangsdiagramme

2.3 Motivation – Komplexe Transformationen an QUEST-Modellen

In diesem Abschnitt wollen wir die Motivation für komplexe Transformationen an Produktmodellen und für die Entwicklung von ODL besprechen. Dazu wollen wir zunächst die Begriffe *einfache Transformation* und *komplexe Transformation* erläutern.

- **Einfache Transformation**
  Unter einer einfachen Transformation verstehen wir eine einzelne Änderungen, die an einem Modell durchgeführt wird – dies könnte das Erstellen, Hinzufügen oder Löschen eines Modellelements sein, das Benennen eines Modellelements, die Zuweisung eines Datentyps zu einem Kanal usw.

- **Komplexe Transformation**
  Eine komplexe Transformation ist eine Abfolge einfacher Transformationen. Beispielsweise wäre eine Operation ”Verbinde zwei Komponenten durch einen Kanal” bereits eine komplexe Transformation, denn sie besteht aus mehreren einfachen Transformationen – einen neuen Kanal erstellen, den Kanal benennen, dem Kanal einen Datentyp zuweisen, einen Ausgangsport der ersten Komponente mit dem Kanal verbinden, einen Eingangsport der zweiten Komponente mit dem Kanal verbinden.

→ Aus den obigen Definitionen sieht man, dass die Entwicklung eines Modells nichts anderes als die Anwendung verschiedener Transformationen ist. Diese Feststellung ist ausschlaggebend für die Entwicklung von Werkzeugen, die die Durchführung von Transformationen für den Benutzer vereinfachen.

Wie wir gesehen haben, besteht schon die relativ einfache Operation ”Verbinde zwei Komponenten durch einen Kanal” aus mehreren einfachen Transformationen. Der Aufwand für solche Operationen wird noch größer, wenn sie mehrmals wiederholt werden müssen, z.B. bei der Operation ”Verbinde zwei Komponenten durch drei Kanäle, die jeweils den Datentyp Int, Float und String haben” – hier müsste der Benutzer ein und dieselbe Abfolge von einfachen Transformationen dreimal wiederholen. Es wäre also von Vorteil, wenn die Abfolge von Transformationen festgelegt werden könnte, sodass der Benutzer nicht mehr jede einzelne einfache Transformation selbst anstoßen müsste, sondern nur die notwendigen Eingaben in einem automatisch ausgeführten Vorgang machen könnte.

Komplexe Transformationen bieten folgende Vorteile:

- **Geringere Fehleranfälligkeit**
  Wenn ein Benutzer die gleiche Transformationsabfolge mehrmals hintereinander ausführen muss, steigt die Wahrscheinlickheit für Fehler, die durch ”Verklicken”, ”Vertippen” und andere Flüchtigkeitsfehler des Benutzers zustande kommen. Nicht zuletzt könnte der Benutzer die Reihenfolge der Transformationen verwechseln, was ebenfalls zu Fehlern führen könnte.

- **Höhere Effizienz**
  Transformationen, die vom Benutzer manuell ausgeführt werden müssen, können wesentlich mehr Zeit in Anspruch nehmen, als gleiche Transformationen, die automatisch ablaufen. Dies ist vor Allem dann der Fall, wenn mehrere gleiche Transformationen ausgeführt werden müssen.

- **Höhere Benutzerfreundlichkeit**
  Die mehrfache Durchführung ein und derselben monotonen Operationsabfolge kann sehr ermüdend auf menschliche Benutzer wirken, insbesondere, wenn jede auszuführende Operation mehrere Mausklicks mit einer eventuell anschließenden Texteingabe benötigt. Eine Automatisierung solcher Abläufe würde für den Benutzer eine wesentliche Erleichterung der Arbeit bedeuten.

Wir wollen die oben aufgeführten Vorteile komplexer Transformationen am Beispiel einer ODL-Abfrage demonstieren, die in Anlehnung an das Beispiel aus [Sch01] auf S.4 alle Ports des Systems mit dem Namen ”out” zu ”outPort” umbenennt:

```
exists p:Port.( /* Iterate over all ports of the system */
  /* Rename port only if the specified condition is fulfilled */
  is Name( p, ”out” ) and result has Name( p, ”outPort” )
)
```

Um dieselbe Arbeit manuell auszuführen, müsste der Benutzer alle Komponenten nach Ports mit dem Namen ”out” durchsuchen und die gefundenen Ports von Hand umbennen – dies könnte, abhängig von der Projektgröße, sehr viel Zeit in Anspruch nehmen und wäre anfällig für Tippfehler bei der Umbenennung. Die Eingabe der obigen ODL-Abfrage mit äquivalenter Wirkung nimmt dagegen weniger als eine Minute in Anspruch, die Ausführungszeit liegt gar im Millisekuden- bis Sekundenbereich.

Im Rahmen des Systementwicklungsprojekts [Tracht] wurde exemplarisch folgende komplexe Transformation implementiert: in einer Komponenten wurde ein Kanalbündel ausgewählt, in das anschließend eine andere Komponente eingefügt wurde – dafür wurden alle Kanäle aus dem Kanalbündel gelöscht und anschließend neue Kanäle erstellt, die Ports des Kanalbündels mit den Ports der eingefügten Komponente verbinden. Die Abbildung 2.5 zeigt, wie das Ergebnis der Anwendung dieser Transformation (aus QUEST in AutoFocus importiert) aussehen könnte.

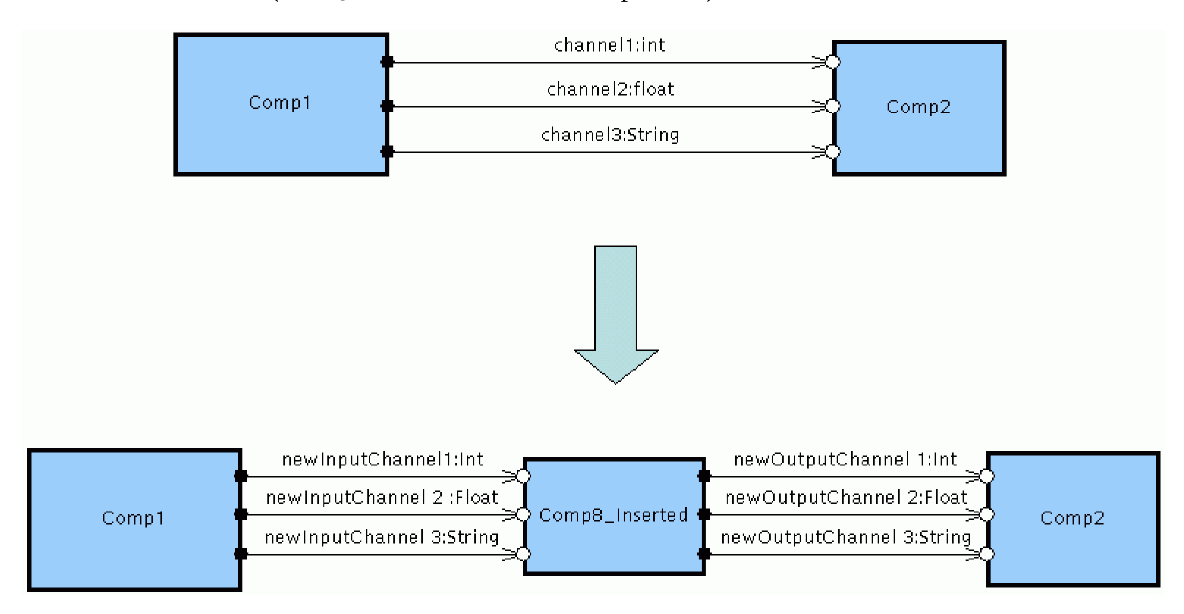


Abbildung 2.5: Komplexe Transformation in QUEST: Kanalbündelauftrennung

Diese beschriebene komplexe Transformation wurde als QUEST-Modul in Java implementiert, wobei nur die Metamodell-Schnittstelle für den Zugriff auf Modelle benutzt wurde. Die Implementierungszeit (Einarbeitung und Entwicklerhandbuch nicht mitgerechnet) betrug circa einen Monat. Damit sehen wir bereits ein Problem bei der Realisierung komplexer Transformationen durch dedizierte Programmmodule – die Realisierung eines neuen Programmmodul erfordert Java-Programmierkenntnisse sowie Kenntnis der QUEST-Metamodell-Schnittstelle, kann mehrere Wochen in Anspruch nehmen und wird nur für die eine implementierte komplexe Transformation einsetzbar sein.

Dies war die Motivation für die Erstellung einer Query-Sprache, die auf dem QUEST-Metamodell operiert und die Beschreibung komplexer Transformationen ermöglicht. Eine solche Query-Sprache bietet mehrere Vorteile:

- **Schnelle Programmierung**
  Operationen können in kurzer Zeit programmiert werden – für die Erstellung einfacher Abfragen wird weniger als eine Minute benötigt.

- **Lesbarkeit**
  Abfragen in einer Query-Sprache sind viel leichter zu lesen als äquivalente Abfragen, die mit Sprachmitteln einer allgemeinen Programmiersprache wie Java oder C++ implementiert wurden.

- **Flexibilität**
  Operationen sind leicht zu modifizieren. Dafür muss lediglich die entsprechende Abfrage angepasst werden, welche anschließend sofort ausgeführt werden kann.

- **Leichte Erlernbarkeit**
  Der Benutzer benötigt keine Programmierkenntnisse in Java und keine Kenntnisse über die QUEST-Metamodell-Schnittstelle. Die Kenntnis der Query-Sprache selbst genügt, um komplexe Transformationen zu programmieren.

- **Garantierte Terminierung**
  Für Abfragen in einer Query-Sprache kann die Terminierung der Berechnung garantiert werden. Dies wird erreicht, indem alle Sprachkonstrukte auf endlichen Universen operieren und die Deklaration rekursiver Terme untersagt ist – die einzige zulässige Form der Rekursion ist die Verwendung des Fixpunktoperators in einer Form, die immer zu Rekursionen endlicher Tiefe führt (beispielsweise Fixpunktoperator auf Mengen, wobei die kleinste oder die größte Teilmenge eines endlichen Basistyps ermittelt werden soll, bei der alle Elemente einer Restriktionsbedingung genügen).

- **Einfache Anpassung an Metamodell-Änderungen**
  Eine Query-Sprache ist sehr einfach an Metamodell-Änderungen anzupassen. Solange das Metamodell der im Abschnitt 3.1 vorgestellten Abstraktion genügt, werden in den meisten Fällen überhaupt keine Änderungen an der Query-Sprache selbst vorzunehmen sein – Änderungen können dann das Auswertungssystem oder die Schnittstelle zum Metamodell betreffen und müssen sich nicht in einer Änderung der Sprachkonstrukte niederschlagen.

- **Sicherheit**
  Eine Query-Sprache kann, im Unterschied zu einer allgemeinen Programmiersprache wie Java oder C++, eng an die mathematische Notation der Prädikatenlogik angelehnt werden, womit ihre Ausdrucksmächtigkeit kontrollierter und überschaubarer wird. Deshalb ist das Risiko unerwünschter Nebeneffekte durch die Transformation eines Modells mithilfe einer Abfrage geringer, als es bei der Programmierung der gleichen Transformation in einer allgemeinen Programmiersprache ist.

Das Konzept einer solchen Query-Sprache für QUEST-Modelle wurde mit der *Operation Definition Language (ODL)* in [Sch01] vorgestellt – das nächste Kapitel behandelt das Konzept und die Grundlagen von ODL sowie die erste Implementierung im Rahmen von [Pasch].

Kapitel 3

Grundlagen von ODL

In diesem Kapitel behandeln wir die Grundlagen von ODL. Der Abschnitt 3.1 befasst sich mit der Konzeption von ODL. Der Abschnitt 3.2 geht auf die erste Implementierung eines ODL-Interpreters für QUEST ein.

3.1 Konzeption

Dieser Abschnitt basiert in weiten Teilen auf [Sch01], wo die *Operation Definition Language (ODL)* vorgestellt wurde.

Als Erstes wollen wir die Formalisierung des Metamodells beschreiben, die für die Definition von ODL Anwendung fand. Ein Metamodell MM besteht aus einem Paar (ME, MR), wobei ME eine Familie von Metamodellentitäten $ME = \{ME_1, \ldots, ME_m\}$ und MR eine Familie von Metamodellrelationen $MR = \{MR_1, \ldots, MR_n\}$ ist. Eine Metamodellrelation ist eine Relation von Metamodellentitäten der Form $MR_i \subseteq ME_{j_1} \times \ldots \times ME_{j_k}$ mit $k > 0$.

Metamodellentitäten sind paarweise disjunkte Mengen von Modellementen – Modellelemente oder auch Entitäten sind damit Instanzen von Metamodellelementen. So wären Komponente oder Port Metamodellelemente, während eine Komponente "Comp1" oder ein Port "Slot1" Entitäten sind.

Der Einfachheit halber werden Attribute von Metamodellentitäten als Spezialfall von Relationen formalisiert – wenn eine Metamodellentität E ein Attribut $attr$ des Typs $type$ hat, so interpretieren wir es als Relation $RE_{attr} \subseteq E \times type$.

In ODL verwenden wir folgende Notation: ein Attribut $attr$ der Entität e wird als $e.attr$ geschrieben; Relationsinstanzen einer Metamodellrelation $R \subseteq E_1 \times \ldots \times E_k$ zwischen Entitäten $e_i \in E_i$ werden als $R(e_1, \ldots, e_k)$ notiert.

ODL-Abfragen operieren auf Produktmodellen, die auf einem Metamodell basieren. In dieser Hinsicht gibt es eine Analogie zwischen SQL und ODL, wobei folgende Entsprechungen zwischen QUEST-Modellen und Datenbanken gelten:

ODL	$\longleftrightarrow$	SQL
Metamodell	$\longleftrightarrow$	Entity-Relationship-Modell
(Produkt-)Modell	$\longleftrightarrow$	Datenbankausprägung

Sowohl SQL als auch ODL sind Query-Sprachen, mit denen man keine Programme schreiben kann, deren Abfragen aber immer terminieren, da sie auf endlichen Universen operieren und keine Schleifen zulassen, die potenziell endlos sein könnten (insbesondere keine WHILE-Schleifen). ODL ist nicht an ein konkretes Metamodell gebunden, sondern kann mit jedem Metamodell verwendet werden, dass der oben dargelegten Formalisierung genügt. Ähnlich dazu ist SQL nicht an ein konkretes Datenbankschema gebunden, sondern wird mit verschiedenen Schemata eingesetzt, die für SQL die Rolle des Metamodells einer Datenbank spielt.

Als Nächstes wollen wir auf den grundlegenden Aufbau von ODL eingehen. Der ODL-Sprachumfang lässt sich in drei Teilmengen unterteilen:

- **Consistency Constraint Language (CCL)**
  Mithilfe der Consistency Constraint Language kann man, ähnlich zu OCL, Konsistenzbedingungen für ein Modell definieren und auswerten, ohne dass durch die Auswertung das Modell verändert werden kann.

  CCL ist im Wesentlichen eine Notation für die Prädikatenlogik erster Stufe auf einem Metamodell. Nehmen wir als Beispiel die Konsistenzbedingung, dass in QUEST-Modellen jede Komponente entweder Unterkomponenten oder einen Automaten besitzen muss, der ihr Verhalten beschreibt. Die prädikatenlogische Notation ist

$$\begin{array}{l} \forall c \in Component.(\\ \quad \exists c_2 \in Component.\ isSubComponents(c, c_2) \vee \\ \quad \exists a \in Automaton.\ isAutomaton(c, a) \\ ) \end{array}$$

  Diese Konsistenzbedingung lässt sich direkt in eine CCL-Abfrage übersetzen:

```
forall c:Component.(
  ( exists c2:Component. is SubComponents( c, c2 ) ) or
  ( exists a:Automaton. is Automaton( c, a ) )
)
```

  Die CCL-Grammatik kann vereinfacht wie folgt beschrieben werden:

  <Formel> ::= not <Formel>
  <Formel> ::= (<Formel>)
  <Formel> ::= <Formel> and <Formel>
  <Formel> ::= <Formel> or <Formel>
  <Formel> ::= <Formel> implies <Formel>
  <Formel> ::= <Formel> equiv <Formel>
  <Formel> ::= forall <Variable>:<Typ> . <Formel>
  <Formel> ::= exists <Variable>:<Typ> . <Formel>

- **Modellmodifikationen**
  Um Änderungen am Modell durchführen zu können, benötigen wir Sprachkonstrukte zum Erstellen von Modellelementen sowie zur Modifikation von Relationen zwischen Modellelementen. Dafür werden drei Sprachkonstrukte eingeführt:

  <Formel> ::= new <Variable>:<Typ> . <Formel>
  <Formel> ::= result <Formel>
  <Formel> ::= result not <Formel>

  Der `new`-Quantor erstellt ein neues Modellelement, ein `result`-Term fügt eine Entität in eine Relation hinzu oder setzt ein Attribut, ein `"result not"`-Term entfernt eine Entität aus einer Relation.

- **Benutzerinteraktionen**
  Zur Durchführung von Benutzereingaben wurde der `context`-Quantor eingeführt:

  <Formel> ::= context <Variable>:<Typ> . <Formel>

Für eine vom `context`-Quantor gebundene Variable wird eine Benutzerabfrage gestartet, in welcher der Benutzer einen Wert für die Variable eingeben kann.

Wir wollen die Benutzerinteraktionen und die Modellmodifikationen am Beispiel einer ODL-Abfrage veranschaulichen, die für eine vom Benutzer ausgewählte Komponente eine Unterkomponente erstellt und dieser einen vom Benutzer eingegebenen Namen zuweist:

```
context c:Component. /* Select a component */
context name:String. /* Enter a name for a new component */
new c2:Component.( /* Create a new component */
  /* Assign the entered name to the new component */
  result has Name( c2, name ) and
  /* Register the new component as a subcomponent of the
     selected component */
  result has SubComponents( c, c2 )
)
```

Die Erweiterung des CCL-Sprachumfangs um die Sprachkonstrukte für Modellmodifikationen und Benutzerinteraktionen ergibt den Sprachumfang von ODL. Wie schon erwähnt, operiert ODL auf endlichen Universen (jedes Modell kann nur endlich viele Entitäten enthalten, die endlich viele Relationen besitzen) und enthält keine Sprachkonstrukte, die eine Endlosschleife ermöglichen würden. Damit ist garantiert, dass eine ODL-Abfrage nach endlicher Zeit terminiert und ein Ergebnis liefert.

An dieser Stelle wollen wir auf Ähnlichkeiten und Unterschiede zwischen ODL und OCL (Object Constraint Language) eingehen.

OCL ist ein Bestandteil des UML-Standards, der zur Spezifikation von Konsistenzbedinungen dient, welche nicht mit anderen UML-Sprachmitteln definiert werden können. OCL wurde entwickelt, um Konsistenzbedinungen auf UML-Modellen in einer Notation definieren zu können, die zwar formell ist, aber an die natürliche Sprache angelehnt ist und daher für Leser ohne mathematischen Hintergrund verständlich bleibt. In dieser Hinsicht dienen CCL und OCL demselben Zweck, nämlich der Spezifikation von Konsistenzbedinungen mithilfe einer leicht zu verstehenden Notation. Wie auch CCL ist OCL keine Programmiersprache und unterstützt keine Programmflusskontrolle. Ein wichtiger Aspekt von OCL ist, dass sie eine reine Spezifikationssprache ist – die Auswertung von OCL-Ausdrücken kann keine Seiteneffekte haben, d.h., ein OCL-Ausdruck kann keine Änderungen am Modell bewirken.

Ausführliche Informationen zu OCL liefert die Sprachspezifikation von OCL 2.0, die in [OCL] zu finden ist. Ein OCL-Interpreter ist in das QUEST-Tool integriert.

Die Möglichkeit von Modellmodifikationen und Benutzerinteraktionen bildet den grundsätzlichen Unterschied zwischen ODL und OCL. ODL kann wie OCL zur Definition von reinen Konsistenzbedingungen benutzt werden, die keine Seiteneffekte haben, indem die ODL-Abfragen auf den Sprachumfang von CCL beschränkt bleiben. Wird der über CCL hinausgehende Sprachumfang von ODL genutzt, so können Operationen definiert werden, die Möglichkeiten zur Veränderung von Modellen und Ausführung von Benutzerinteraktionen bieten und damit Transformationen von Modellen ermöglichen.

Weitere Informationen zur Konzeption und Grundlagen von ODL liefert [Sch01].

3.2 Erste Implementierung des ODL-Interpreters

Die erste Implementierung eines ODL-Interpreters für QUEST wurde im Rahmen von [Pasch] erstellt. Die ODL-Abfragen können in einem Editorfenster eingegeben werden, die Ergebnisse der

Auswertung werden wahlweise in einem separaten Ausgabefenster oder im Editorfenster angezeigt (Abbildung 3.1). Die Architektur des ODL-Systems und der implementierte Sprachumfang werden ausführlich in [Pasch] beschrieben. Wir wollen daher nur kurz den implementierten Sprachumfang beschreiben, der den Ausgangspunkt für die Erweiterungen darstellt.

Abbildung 3.1: ODL-Editorfenster

Der Sprachumfang der ersten ODL-Interpreter-Realisierung entspricht der in [Sch01] (S.16-17) spezifizierten Grammatik mit der Einschränkung, dass Selektorausdrücke und Funktionsausdrücke nicht unterstützt wurden. Damit standen folgende Sprachkonstrukte zur Verfügung:

- **Datentypen**
  Das Typsystem enthielt zwei Gruppen von Datentypen: die Grundtypen `Boolean`, `Int` und `String` sowie Metamodelltypen, die alle vom Metamodell definierten Entitäten darstellen – beispielsweise entpricht der Metamodellentität ”Port” der ODL-Datentyp `Port`.

- **Quantifizierungen**
  Variablen von Typen mit endlicher Domäne konnten mit dem Universalquantor `forall` und dem Existenzquantor `exists` quantifiziert werden. Außerdem konnten mit dem `new`-Quantor neue Entitäten erstellt werden.

  Der `context`-Quantor, der für die gebundene Variable eine Benutzereingabe starten soll, wurde in den Sprachumfang intergriert, ein entsprechendes ODL-Query-Subsystem war allerdings nicht Bestandteil der Implementierung, sodass von `context`-Quantoren gebundene Variablen mit festen Werten (z.B. `null` für Entitäten oder `0` für ganze Zahlen) belegt wurden.

  Eine Quantifizierung besteht aus dem Quantor, der gebundenen Variablen, dem Variablentyp und dem quantifizierten Term. Beispiel:
  ```
  forall c1:Component. exists c2:Component. c1 = c2
  ```

- **Logische Operationen**
  Zwei ODL-Propositionen konnten mit den Junktionen `and`, `or`, `implies` und `equiv` verknüpft werden. Mit dem einstelligen `neg`-Operator konnte eine ODL-Proposition negiert werden. Beispiel:
  ```
  forall a:Boolean. forall b:Boolean.(
    a = b equiv exists c:Boolean.( a = c and b = c ) )
  ```

- **Konstanten**
  Für die Grundtypen `Boolean`, `Int` und `String` konnte jeder zulässige Wert als Konstante eingegeben und in ODL-Abfragen verwendet werden. Beispiel:
  ```
  (10 = 10 and ”ABC” = ”ABC”) equiv true
  ```
  Die Eingabe einer Entität als Konstante ist nicht möglich, da im Allgemeinen nicht bekannt ist, aus welchen Werten eine Entität besteht und auf welche Weise sie eindeutig identifiziert werden kann (der Name und Typ genügen als Identifikation oft nicht). Entitätskonstanten sind auch nicht notwendig, da zur Erstellung neuer Entitäten der `new`-Quantor dient und der Zugriff auf eine existierende Entität mit der Quantifikation `exists var:EntityType. Condition` erfolgt, wobei `Condition` eine Bedingung definieren soll, die die gesuchte Entität eindeutig identifiziert.

- **Gleichheitstest**
  Zwei Werte vom gleichen Typ können auf Gleichheit getestet werden. Für die Grundtypen `Boolean`, `Int` und `String` wird dabei einfach auf Wertegleichheit getestet. Zwei Entitätsvariablen dagegen werden nur dann als gleich angesehen, wenn sie ein und dieselbe Entität referenzieren – zwei verschiedene Entitäten, die den gleichen Namen tragen und die gleichen Relationen aufweisen, werden als ungleich betrachtet. Nehmen wir zwei ODL-Abfragen als Beispiele:
  Test auf Wertgleichheit:
  ```
  exists b:Boolean. b = true
  ```
  Ungleichheit von Entitäten bei gleichem Namen:
  ```
  exists p1:Port. exists p2:Port.(
    is Name(p1, ”out”) and is Name(p2, ”out”) and neg p1 = p2 )
  ```

- **Relationstest**
  Mit einem Term der Form `is SomeRelation( entity, object )` kann getestet werden, ob die Relation `SomeRelation` zwischen der spezifizierten Entitäten und dem spezifizierten Objekt besteht. Dabei ist `object` eine andere Entität (wenn `SomeRelation` eine Assoziation ist) oder ein Wert (wenn `SomeRelation` ein Attribut ist). Betrachten wir Beispiele für verschiedene Relationen:
  Relation ist ein Attribut:
  ```
  exists c:Component. is Name( c, ”Centrale” )
  ```
  Relation ist eine Assoziation:
  ```
  exists comp:Component. exists subComp:Component.
    is SubComponents( comp, subComp )
  ```
  Wie wir sehen, wird in beiden Fällen wird dieselbe Syntax verwendet.

  An dieser Stelle muss darauf hingewiesen werden, dass in der ersten ODL-Implementierung Relationen in der Form `isSomeRelation( entity, object )` notiert wurden. Im Laufe der Erweiterung des ODL-Auswertungssystems wurde die Notation verändert, sodass nun `is` und (für Relationsmodifikationen) `has` selbständige Schlüsselwörter sind, die durch ein Leerzeichen vom Namen der Relation getrennt werden. Folgende ODL-Abfragen stellen Beispiele für die früher benutzte Notation und die aktuelle Notation dar:
  Frühere Notation: `exists c:Component. isName( c, ”Centrale” )`
  Aktuelle Notation: `exists c:Component. is Name( c, ”Centrale” )`

- **Relationsmodifikationen**
  Neben der Möglichkeit zu überprüfen, ob eine Relation zwischen zwei Entitäten bzw. einer Entität und einem Wert besteht, wurde die Möglichkeit implementiert, Relationen zu modifizieren. Hierbei muss zwischen der Erstellung einer Relation und dem Löschen einer Relation unterschieden werden – die entsprechenden ODL-Terme lauten:

- `result has SomeRelation( entity, object )`
  für die Erstellung oder Veränderung einer Relation
- `result not has SomeRelation( entity, object )`
  für das Löschen einer Relation

Die Veränderung bzw. das Löschen einer Relation hat, abhängig von der Art der Relation, folgende Auswirkungen:

- Für eine einwertige Assoziationen, d.h., eine Assoziation zu höchstens einer anderen Entität, wird die Entität `object` zu der Entität `entity` assoziiert bzw. wird die Assoziation gelöscht.
- Für eine mehrwertige Assoziation wird die Entität `object` in die Kollektion der zu `entity` assoziierten Entitäten eingetragen bzw. aus dieser Kollektion entfernt.
- Wenn `SomeRelation` ein Attribut ist, so wird der Wert `object` als Attributwert für `SomeRelation` gespeichert. Das Löschen eines Attributs hat keine Auswirkung – der Attributwert bleibt unverändert.

Zur Veranschaulichung wollen wir wir folgende Beispiele für Relationsmodifikationen betrachten:

- Attributtest und Attributzuweisung:
  ```
  exists p:Port.( is Name( p, ”out” ) and
    result has Name( p, ”outPort” ) )
  ```
  Hier werden alle Ports mit dem Namen `”out”` zu `”outPort”` umbenannt.
- Test, Erstellen und Löschen einer Assoziation:

  Mit der Abfrage
  ```
  context c1:Component. context c2:Component.(
    exists c:Component.( is SubComponents( c1, c ) and
      result not has SubComponents( c1, c ) and
      result has SubComponents( c2, c ) ) )
  ```
  werden für zwei vom Benutzer ausgewählte Komponenten alle Unterkomponenten aus der ersten Komponente in die zweite verschoben.
- Einfügen einer neuen Entität ins Modell:

  Um eine Entität ins Modell einzufügen, muss sie mit dem `new`-Quantor erstellt und anschließend einer Eigentümer-Komponente zugewiesen werden. Betrachten wir als Beispiel das Einfügen einer neuen Komponente:
  ```
  context ownerComp:Component. context name:String.
    new comp:Component.(
      result has Name( comp, name ) and
      result has SubComponents( ownerComp, comp ) )
  ```
  Der Benutzer wählt eine Komponente, in welche die neue Komponente als Unterkomponente eingefügt werden soll, und gibt den Namen für die neue Komponente ein. Anschließend wird eine neue Komponente erstellt, ihr wird der vom Benutzer eingegebene Name zugewiesen und schließlich wird sie in die Kollektion der Unterkomponenten der spezifizierten Eigentümer-Komponente eingefügt.

Weitere Details zur ersten Implementierung des ODL-Interpreters finden sich in [Pasch].

Kapitel 4

Erweiterung von ODL

Zur Aufgabe der Diplomarbeit gehörte die Erweiterung des Sprachumfangs von ODL sowie die Konzeption und Implementierung einer interaktiven Benutzerschnittstelle für die Auswertung von ODL-Abfragen.

In den folgenden Abschnitten erläutern wir im Einzelnen, welche Erweiterungen am Sprachumfang von ODL vorgenommen werden sollten und wie die Benutzerschnittstelle konzipiert wurde.

4.1 Erweiterung des Sprachumfangs

Wie im Abschnitt 3.2 erwähnt, waren einige wichtige Sprachkonstrukte von ODL im implementierten Sprachumfang noch nicht enthalten. In [Sch01] (S. 5, S. 16-19) wurden mehrere Erweiterungen des ODL-Sprachumfangs vorgestellt, von denen die folgenden im Rahmen der vorliegenden Diplomarbeit zu realisieren waren:

- **Produkttypen**
  Bislang konnten nur Variablen unärer Typen deklariert werden, z.B. `var:Boolean` oder `var:Component`. Jetzt sollte die Deklaration von Variablen ermöglicht werden, deren Typ ein Tupel aus mehreren Typen ist:

  `var:(ident1:type1,ident2:type2,...,ident_n:type_n)` (4.1)

  Damit wären beispielsweise folgende Typdeklarationen möglich:

  - `var:(comp:Component, port:Port)`
  - `var:(comp:Component, name:String, ports:(p1:Port,p2:Port))`

  Hierbei darf jeder gültige ODL-Typ als Elementtyp eines Produkttyps verwendet werden.

- **Selektoren**
  Für den Zugriff auf Elemente eines Produkttyps und auf Attribute bzw. Assoziationen von Modellelementen werden Selektoren verwendet. Der Zugriff auf ein Element bzw. Attribut einer Variablen erfolgt über das Anhängen eines Punkts und des Selektors an die Variable:

  `var.selector` (4.2)

  Betrachten wir Beispiele für den Zugriff auf Elemente eines Produkttyps:

  - `exists ports:( p1:Port, p2:Port ). ports.p1 = ports.p2`
    liefert alle Paare gleicher Ports

- `exists var:( ch:Channel, ports:( in:Port, out:Port ) ).( is SourcePort( var.ch, var.ports.out ) and is DestinationPort( var.ch, var.ports.in ) )`
  liefert alle Tupel aus einem Kanal und einem Portpaar, bei denen das Portpaar aus dem Eingangsport und dem Ausgangsport des Kanals besteht.

Und nun einige Beispiel für den Zugriff auf Attribute und Assoziationen von Modellelementen:

- `exists c:Component. c.Name = ”Comp1”`
  liefert alle Komponenten mit dem Namen `”Comp1”` (diese Abfrage ist äquivalent zur Abfrage `exists c:Component. is Name( c, ”Comp1” )`)
- `context c:Component.( new p:Port.( result has Name( p, c.Name ) and result has Ports( c, p ) )`
  lässt den Benutzer eine Komponente auswählen und fügt bei ihr einen neuen Port hinzu, dessen Name gleich dem Komponentennamen ist. Im Unterschied zur vorherigen Abfrage kann diese nicht mehr in eine gültige äquivalente Abfrage umformuliert werden, die keinen Gebrauch vom Attributzugriff über Selektoren macht[1].
- `exists p:Port. p.Type.Text = ”Int”`
  findet alle Ports mit dem Datentyp ”Int”

Das nächste Beispiel zeigt den Zugriff auf Produkttyp-Elemente und Modellelement-Attribute innerhalb ein und desselben Selektorausdrucks:

- `exists ports:( p1:Port, p2:Port ).( ports.p1.Type.Text = ports.p2.Type.Text )`
  liefert alle Portpaare, die aus Ports gleichen Datentyps bestehen.

• **Eingeschränkte Typen** (`RestrictedType`, in [Sch01] (S. 18) als Mengenkomprehension bezeichnet)
  Oft ist es notwendig, anstatt aller Instanzen eines Typs nur diejenigen zu betrachten, die einer bestimmten Bedingung genügen. Zu diesem Zweck werden eingeschränkte Typen eingeführt. Die Deklaration eines eingeschränkten Typs ist an die mathematische Schreibweise für Teilmengen einer Basismenge angelehnt, wobei hier Typen die Rolle von Mengen spielen.

```
var:{ident:base_type | restriction_term über ident}        (4.3)
```

Als Basistyp eines eingeschränkten Typs darf jeder gültige ODL-Typ dienen. Der Restriktionsterm definiert, welche Instanzen des Basistyps in dem eingeschränkten Typ enthalten sind – er darf die lokale Variable `ident` und andere im Namensraum bereits deklarierte Variablen verwenden. Zudem darf der Restriktionsterm nur eine CCL-Proposition sein, d.h., dass Schlüsselwörter `result`, `not result` sowie Quantoren `new` und `context` nicht zugelassen sind – damit wird sichergestellt, dass ein Restriktionsterm keine Änderungen am Modell vornehmen kann und keiner Benutzerinteraktionen bedarf.

Bespiele:

- `var:{ p:Port | true }`
  Alle Ports
- `var:{ comp:Component | comp.Name = ”Comp1” }`
  Alle Komponenten mit dem Namen `”Comp1”`.

[1] Die semantisch äquivalente Abfrage `context c:Component. new p:Port. exists s:String.( is Name( c, s ) and result has Name( p, s ) and result has Ports( c, p ) )` kann nicht ausgewertet werden, weil eine Iteration über Instanzen des unendlichen Typs `String` nicht möglich ist.

- `var:{ comps:( c1:Component, c2:Component ) |`
  `is SubComponents( comps.c1, comps.c2 }`
  Alle Komponentenpaare, bei denen die zweite Komponente eine Unterkomponente der ersten ist.

- **Mengen**
  Bislang hatte jede Variable als Wert genau eine Instanz ihres Typs. Mit Mengenvariablen wird die Möglichkeit gegeben, mehrere Instanzen eines Typ in einer Variable zu speichern:

  `var:set base_type` (4.4)

  Eine Mengenvariable ist eine Teilmenge des Basistyps – man kann sie als Kollektion von Werten des Basistyps auffassen, deren Elemente alle paarweise verschieden sind.

  Um auf die Elemente einer Menge zugreifen zu können, wird das Schlüsselwort `element` eingeführt:

  `setVar:set base_type. exists var:element setVar` (4.5)

  Dabei kann als Menge nicht nur eine Mengenvariable, sondern jeder Ausdruck verwendet werden, dessen Ergebnis eine Menge ist. Die Abfrage

  ```
  exists comp:Component. exists subComp:
  element(comp.SubComponents)
  ```
  (4.6)

  iteriert beispielsweise für jede Komponente über die Menge ihrer Unterkomponenten.

  Mit dem vorangestellten Schlüsselwort `element` agiert jede Mengenvariable oder Ausdruck mit mengenwertigem Ergebnis als Typ, dessen Instanzen genau die in der Menge enthaltenen Elemente sind. In dieser Form können Mengen als Typen für Variablendeklaration in allen ODL-Sprachkonstrukten außer dem `new`-Quantor benutzt werden – solche Typen werden wir als *eingeführte Typen* bezeichnen. Ein mengenwertiger Ausdruck muss bei der Verwendung mit dem Schlüsselwort `element` geklammert werden.

  Hier einige Beispiele für den Einsatz von Mengen:

  - `context setVar:set Component. true`
    Eingabe einer Menge von Komponenten durch den Benutzer.
  - `exists boolSet:set Boolean. true`
    Alle Teilmengen des Typs `Boolean`. Das Ergebnis der Abfrage ist
    `( {}, {false}, {true}, {false,true} )`
  - `exists boolSet:set Boolean.`
    `exists bool:element boolSet. bool = true`
    Alle Teilmengen von `Boolean`, die den Wert `true` enthalten. Das Ergebnis der Abfrage ist
    `( {true}, {false,true} )`
  - `exists comp:Component.`
    `exists subComp:element( c.SubComponents ).`
    `subComp.Name = ”Comp1”`
    Für jede Komponente wird nach einer Unterkomponente mit dem Namen `”Comp1”` gesucht.
  - `context comps:(c1:Component,c2:Component).`
    `exists ports:( p1:element( comps.c1.Ports ),`

```
p2:element( comps.c2.Ports ) ). true
```

Der Benutzer muss ein Komponentenpaar auswählen, wonach alle Portpaare ausgegeben werden, in denen der erste Port zur ersten Komponenten und der zweite Port zur zweiten Komponente gehört.

- **Benamte Prädikate**
  Ein benamtes Prädikat ist als eine Analogie zu Funktionen zu verstehen: es wird mit einem eindeutigen Namen deklariert und kann später in anderen Ausdrücken aufgerufen werden.

  ```
  predicateName( param1:type1,...,param_n:type_n ) :=
  CCL-Proposition über param1,...,param_n
  ```
  (4.7)

  Wie bereits für Restriktionsterme bei eingeschränkten Typen darf die CCL-Proposition Operationen `result`, `not result` sowie Quantoren `new` und `context` nicht enthalten. Außerdem sind Rekursionen innerhalb benamter Prädikate unzulässig: sowohl direkte Rekursionen der Form ”term1 ruft term1 auf”, als auch indirekte Rekursionen der Form ”term1 ruft term2 und term2 ruft term1 auf” führen zu einem Fehler bei der Übersetzung der benamten Prädikate. Zum Aufruf eines bereits definierten benamten Prädikats aus einer anderen ODL-Abfrage heraus wird das Schlüsselwort `call` benutzt.

  Hier einige Beispiele:

  - ```
    nameCondition( p:Port ) := is Name( p, ”Name1” ) or
      is Name( p, ”Name2” ) or is Name( p, ”Name3” )
    ```
    Nun kann man bequem die Abfrage
    ```
      exists ports:(p1:Port, p2:Port).(
        call nameCondition( ports.p1 ) and
        call nameCondition( ports.p2 ) )
    ```
    formulieren.
  - ```
    containsTrue( boolSet:set Boolean ) :=
      exists b:element boolSet. b = true
    ```
    Die Abfrage
    ```
      exists bSet:set Boolean. call containsTrue( bSet )
    ```
    gibt nun alle Teilmengen von `Boolean` zurück, die den Wert `true` enthalten.

  Und schließlich noch ein komplexeres Beispiel. Das benamte Prädikat
  ```
  componentsConnected( c1:Component, c2:Component ) :=
    exists ports:(p1:element(c1.Ports), p2:element(c2.Ports)).(
      (exists ch1:element(ports.p1.OutChannels).
        ch1 = ports.p2.InChannel) or
      (exists ch2:element(ports.p2.OutChannels).
        ch2 = ports.p1.InChannel) )
  ```
  stellt fest, ob zwei Komponenten durch einen Kanal verbunden sind. Die Abfrage
  ```
  exists pair:( c1:Component, c2:Component ).
    call componentsConnected( pair.c1, pair.c2 )
  ```
  gibt alle Paare von Komponenten zurück, die durch mindestens einen Kanal verbunden sind.

Zusätzlich zu den oben angegebenen Sprachkonstrukten wurde ODL um weitere Sprachkonstrukte und Fähigkeiten erweitert:

- **Vergleiche**
  Bisher stand nur der Test auf Gleichheit zweier Werte zur Verfügung. Der Typ der Werte durfte beliebig sein – nur mussten beide Werte, natürlich, typkompatibel sein. Für die Typen, auf

denen eine totale Ordnung existiert (in ODL sind es `String` und `Int`), wurden nun die Vergleichsoperatoren '$<$', '$>$', '$\leq$' sowie '$\geq$' eingeführt. Für den Typ `String` wird hierbei die lexikographische Ordnung herangezogen.

Beispiele:

Ausdruck	Ergebnis
`10 > 5`	`true`
`4 >= 10`	`false`
`"stringA" <= "string"`	`false`
`"stringA" < "stringB"`	`true`

- **Arithmetische Operationen**
  Für den Umgang mit ganzen Zahlen wurden Addition, Subtraktion und Multiplikation eingeführt. Da das Ergebnis einer ODL-Abfrage stets boolesch ist, dürfen arithmetische Ausdrücke nur innerhalb boolescher Terme auftreten. Das bedeutet in der Praxis, dass arithmetische Ausdrücke fast nur innerhalb von Vergleichen und Gleichheiten Anwendung finden, weil das aktuelle QUEST-Metamodell so gut wie keine ganzzahligen Attribute bei Entitäten aufweist, die mit einer Zuweisung der Form `result has SomeNumAttr( entity, 10 )` verändert werden könnten.

  Beispiele:

Ausdruck	Ergebnis
`3 + 5 = 8`	`true`
`3 * ( 10 + 5 ) > 45`	`false`
`2 - 1 * 3 < 0`	`true`
`(-1)*5 + 3*(7-2) = 10`	`true`

  Damit ODL-Abfragen immer erfolgreich ausgewertet werden können, wurde die Division nicht eingeführt, weil hierbei eine Division durch Null und damit ein undefiniertes Ergebnis im Laufe der Auswertung eintreten kann. Auf die Möglichkeiten zur Einführung der Division in ODL wird noch einmal im Abschnitt 6.1.4 eingegangen.

- **Mengenoperationen**
  Zurzeit sind zwei Operationen auf Mengen verfügbar:

  - Leerheitstest: `isEmpty(setVar)`
    Für eine Mengenvariable `setVar` wird mit dem Ausdruck `isEmpty(setVar)` getestet, ob die von der Variablen repräsentierte Menge leer ist. Falls ja, evaluiert der Ausdruck zu `true`, sonst zu `false`.
    Beispiel:
    ```
    exists c:Component. neg isEmpty( c.Ports )
    ```
    liefert alle Komponenten, die mindestens einen Port haben.

  - Mengengröße: `size(setVar)`
    Für eine Mengenvariable `setVar` gibt Ausdruck `size(setVar)` die Anzahl der Elemente in der Menge zurück.
    Beispiel:
    ```
    exists c:Component. size( c.SubComponents ) >= 3
    ```
    liefert alle Komponenten, die mindestens drei Unterkomponenten haben.

  Vorschläge für weitere Mengenoperationen finden sich im Abschnitt 6.1.2.

- **Erweiterte Syntax des `context`-Quantors**
  Die Syntax des `context`-Quantors wurde so erweitert, dass der Benutzer über einen optionalen Parameter einen Texthinweis spezifizieren kann, der im Eingabedialog für die vom `context`-Quantor gebundene Variable angezeigt werden kann.

```
context [ hint = ”A hint message” ] var:type
```
(4.8)

Eine solche Abfrage ist semantisch äquivalent zu der Abfrage `context var:type`. Der einzige Unterschied besteht darin, dass im Eingabedialog für die Variable `var` der Text `”A hint message”` angezeigt wird.

Soll ein mehrzeiliger Text angezeigt werden, müssen die Zeilen durch Kommata getrennt werden:

```
context [ hint = ”First line”, ”Second line” ] var:type
```

Sowohl die Verwendung eines optionales Parameters, als auch der Parameter selbst sind optional – alle folgenden ODL-Abfragen sind damit gültig und semantisch äquivalent:

```
context var:Int. true
context [] var:Int. true
context [ hint = ”A message” ] var:Int. true
context [ hint = ”Line 1”, ”Line 2” ] var:Int. true
```

Zusammenfassend führen wir noch einmal alle Erweiterungen des Sprachumfangs von ODL in der Tabelle 4.1 auf. Im Anhang B wird eine aktualisierte ODL-Grammatik angegeben, die durch die Erweiterung der Grammatik aus [Sch01] um die neuen Sprachkonstrukte entstand.

Bezeichnung	**Erläuterung**
Produkttypen	Typen bestehend aus mehreren Elementen anderer Typen.
Selektoren	Zugriff auf Elemente von Produkttypen sowie auf Attribute und Assoziationen von Modellelementen.
Eingeschränkte Typen	Restriktion eines Basistyps: enthält nur diejenigen Werte des Basistyps, die eine Restriktionsbedingung erfüllten.
Mengen	Mengenvariablen, die Kollektionen von Werten des Basistyps der Menge darstellen; Möglichkeit der Iteration über die Elemente einer Menge.
Benamte Prädikate	Deklaration von CCL-Propositionen, die später aus anderen ODL-Abfragen aufgerufen werden können.
Vergleiche	Vergleichsoperatoren für Zahlen und Strings.
Arithmetische Operationen	Addition, Subtraktion und Multiplikation von Zahlen.
Mengenoperationen	Operationen `isEmpty` und `size` auf Mengen.
Erweiterte Syntax des `context`-Quantors	Für einen `context`-Quantor kann optional ein Hinweistext für den Eingabedialog spezifiziert werden.

Tabelle 4.1: Erweiterung des Sprachumfangs von ODL

4.2 Interaktive Benutzerschnittstelle

Eine allgemeine Beschreibung des ODL-Editors ist in [Pasch] (S. 41-46) gegeben. Deshalb werden wir den Schwerpunkt dieses Abschnitt auf die die Neuerungen legen, die sich im Wesentlichen auf die implementierte Benutzerschnittstelle zu Eingabe von Variablenwerten während einer ODL-Abfrage beziehen.

Eine ODL-Abfrage bedarf während der Auswertung keiner Benutzerinteraktion, solange sie keine `context`-Quantoren enthält. Ein `context`-Quantor startet eine Benutzereingabe für die vom

Quantor gebundene Variable (s. auch [Sch01], S. 5). Das Query-System ist so konzipiert, dass prinzipiell jeder Weg für die Eingabe implementiert werden kann. In der vorliegenden Implementierung werden alle Benutzereingaben in Dialogfenstern durchgeführt. Es ist jedoch genauso möglich, dass beispielsweise Modellelemente nicht aus einer Liste im Dialogfenster, sondern im QUEST-Projektbrowser oder in einem graphischen Editor ausgewählt werden.

Die folgenden Unterabschnitte stellen ein Benutzerhandbuch für die Eingabe von Variablenwerten während einer ODL-Abfrage dar. Sie beschreiben die Bedienung von Eingabedialogen sowie allgemeine Konfiguration des Query-Systems.

4.2.1 Eingabedialoge

Alle Benutzereingaben finden innerhalb eines *Eingabedialogs* statt. Betrachten wir als Beispiel den Eingabedialog, der bei der Auswertung der ODL-Abfrage `context str:String. str="abc"` angezeigt wird (Abb. 4.1). Der Eingabedialog besteht aus drei Teilen:

1) Eingabepanel
In dem Eingabebereich gibt der Benutzer den Variablenwert ein. Der eingesetzte Eingabebereich hängt vom Typ der abgefragten Variablen und von den Einstellungen für Eingabedialoge ab, auf die wir im Abschnitt 4.2.7 näher eingehen werden.

2) Navigationsleiste
Die Navigationsleiste stellt Navigationsbuttons zur Verfügung, mit deren Hilfe der Benutzer zwischen den Eingabedialogen für verschiedene Variablen navigieren kann. Folgende Buttons werden verwendet:

- *Previous*-Button: falls vor der aktuellen Variablen bereits andere eingegeben wurden, kann der Benutzer mithilfe dieses Buttons zur Eingabe der vorherigen Variablen zurückkehren – hierbei geht der Variablenwert aus dem aktuellen Eingabedialog verloren. Wurden bisher keine anderen Variablen eingegeben, so ist der Button deaktiviert.
- *Cancel*-Button: dieser Button bricht die Variableneingabe und damit die Auswertung der gesamten ODL-Abfrage ab.
- *Next*-Button: mit diesem Button schließt der Benutzer die Eingabe einer Variablen ab. Falls weitere Variablen einzugeben sind, so wird zu ihrer Eingabe übergegangen. Dieser Button ist nur dann aktiviert, wenn der Benutzer einen zulässigen Wert für die aktuelle Variable eingegeben hat. Hierbei wird ein Wert als zulässig betrachtet, wenn er einen gültigen Wert für den abgefragten Datentyp darstellt (bei eingeschränkten Typen muss der eingegebene Wert auch die Restriktionsbedingung erfüllen).

Optional wird in der Navigationsleiste ein Hinweistext angezeigt, der als optionaler Parameter dem `context`-Quantor übergeben werden kann. Die Abfrage

```
context [ hint = "Enter a string" ] str:String. str = "abc"
```

ist äquivalent zu der Abfrage

```
context str:String. str="abc"
```

zeigt aber zusätzlich den Hinweis `"Enter a string"` im Eingabedialog an (Abb. 4.2).

3) Werteanzeige
Der Werteanzeige-Bereich dient dazu, die Belegungen bereits bekannter Variablen anzuzeigen, damit der Benutzer informiert ist, in welchem Kontext der Wert für die aktuelle Variable einzugeben ist. Angezeigt werden die Werte von Variablen, die von vorhergehenden `context`-Quantoren gebunden werden und deshalb bereits eingegeben wurden, sowie Werte von Variablen, die von vorhergehenden `forall`-Quantoren gebunden werden.

Die Abfrage `context str1:String.context str2:String.true` zeigt im Eingabedialog für die Variable `str2` im Werteanzeige-Bereich den vorher eingegebenen Wert `"abc"` für `str1` an (Abb. 4.3).

Für die Werteanzeige kann zurzeit entweder ein Textbereich (Abb. 4.3) oder eine Tabelle (Abb. 4.6) eingesetzt werden – auf die entsprechende Einstellung werden wir im Abschnitt 4.2.7 eingehen.

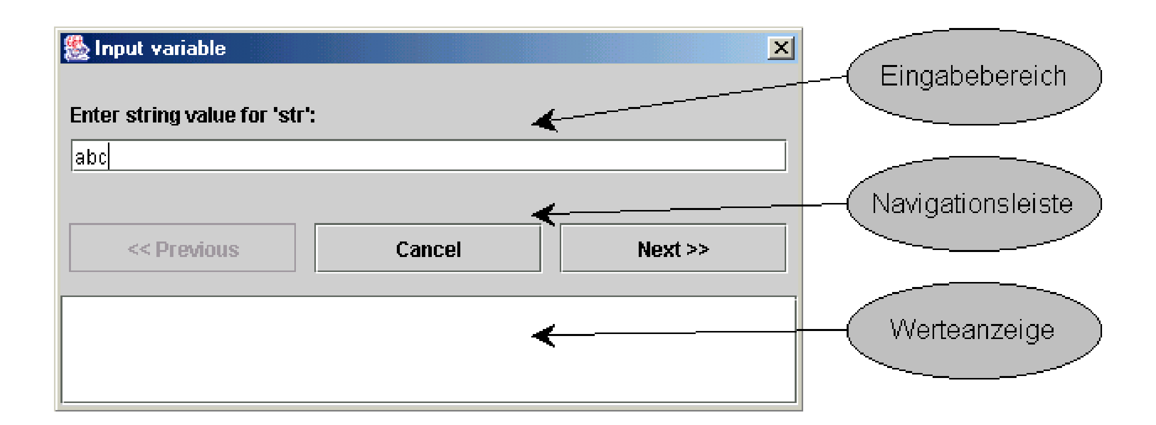


Abbildung 4.1: Eingabedialog

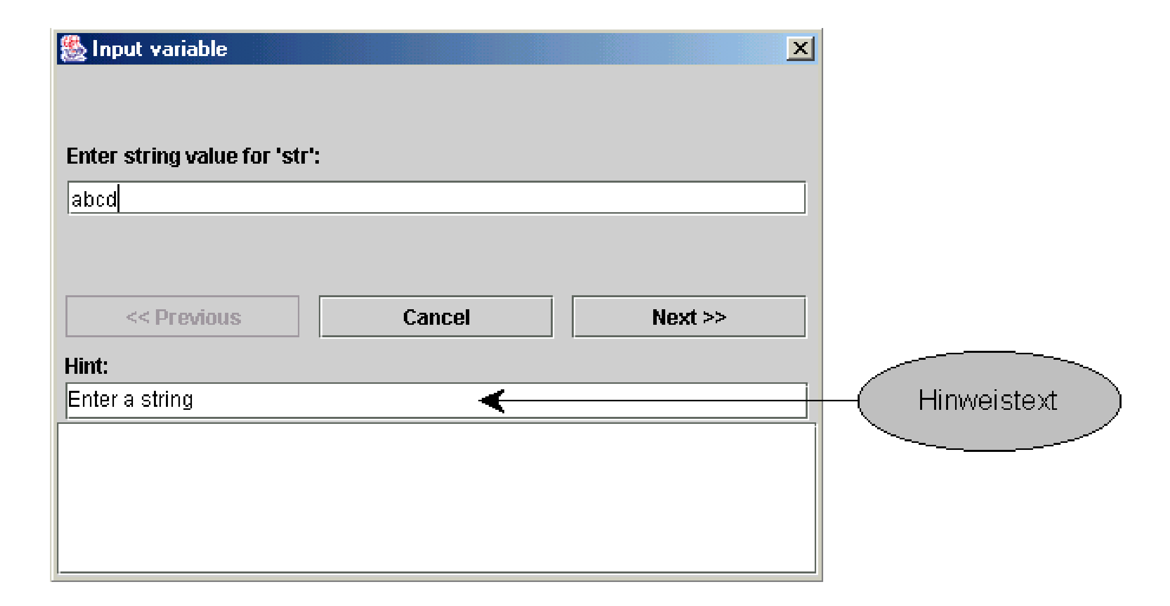


Abbildung 4.2: Eingabedialog mit Hinweistext

Um die Eingabe eines Werts abzuschließen, kann der Benutzer entweder den *Next*-Button betätigen oder auch die Eingabe-Taste im Eingabepanel drücken. Bei der Auswahl eines Werts aus einer Liste ist auch die Bestätigung über einen Doppelklick auf den ausgewählten Wert möglich.

Neben dem bereits vorgestellen Eingabedialog kann ein weiterer Eingabedialog verwendet werden, der sich darin unterscheidet, dass die Größe der drei Dialogbereiche verändert werden kann (Abb. 4.4). Der Einsatz dieses Dialogs ist insbesondere dann sinnvoll, wenn das Dialogfenster insgesamt vergrößert werden muss (z.B. bei der Eingabe komplexer Datentypen), denn in diesem Fall würden im einfachen Dialogfenster alle drei Dialogbereiche proportional zu ihrer ursprünglichen Größe vergrößert, womit der Navigationsbereich und die Werteanzeige gegebenenfalls unnötigen Platz verbrauchten, während der Eingabebereich zu klein ausfiele.

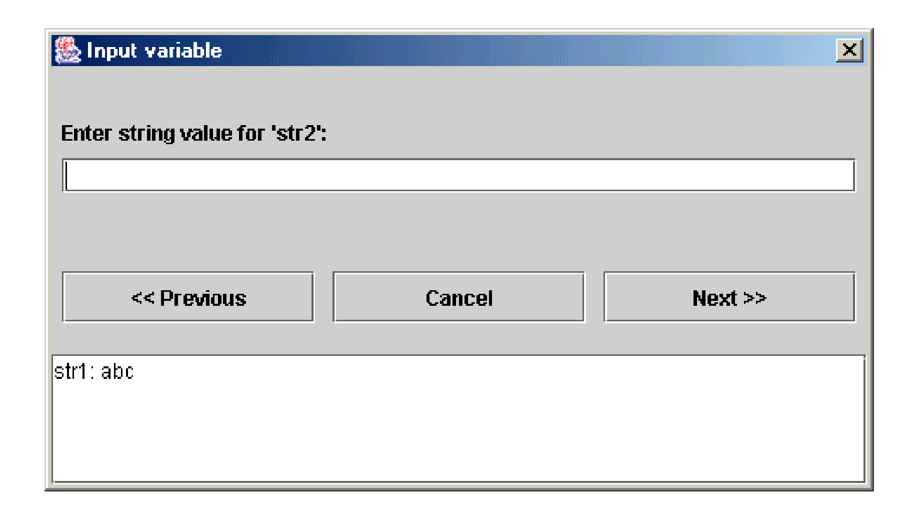


Abbildung 4.3: Eingabedialog mit Werteanzeige

Die Einstellung, welcher der verfügbaren Eingabedialoge für verschiedene Datentypen verwendet werden soll, kann im Konfigurationsdialog für das ODL-Query-Subsysten individuell für jeden ODL-Datentyp vorgenommen werden (Abschnitt 4.2.7).

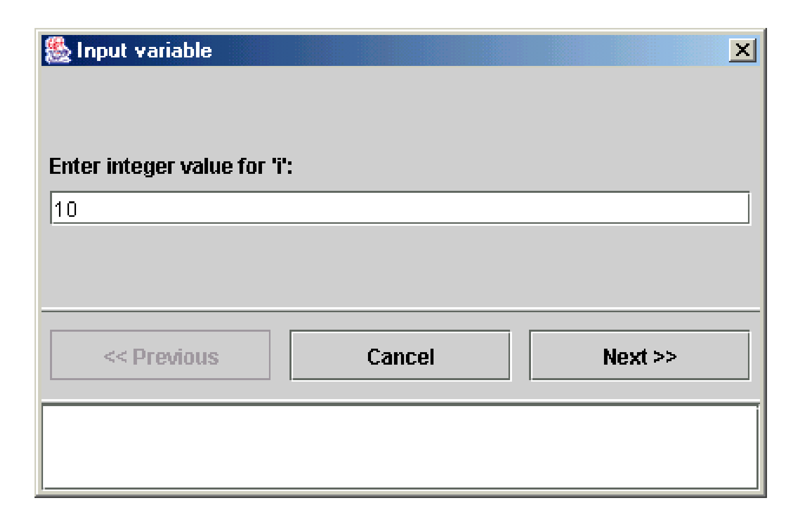


Abbildung 4.4: Eingabedialog mit veränderbaren Bereichsgrößen

4.2.2 Eingabe unärer Werte

Für jeden der unären Datentypen `String`, `Int`, `Boolean` sowie für Metamodelltypen stehen Eingabepanels zur Verfügung, in denen in einem Eingabedialog Variablenwerte eingegeben werden können. Wir beschreiben nun die Eingabemöglichkeiten für unäre Datentypen im Einzelnen:

- **String**
  `String`-Werte werden in einem Textfeld eingegeben (Abb. 4.1). Dabei wird jede Eingabe akzeptiert.

- **Int**
  Ganzzahlige Werte werden ebenfalls in einem Textfeld eingegeben (Abb. 4.4). Es werden nur Eingaben akzeptiert, die eine gültige ganze Zahl darstellen.

- **Boolean**
  Für `Boolean`-Werte stehen mehrere Eingabepanels zur Auswahl: zusätzlich zu dem Textfeld, dass die Eingaben `"true"` und `"false"` akzeptiert, kann ein boolescher Wert mit einem Radiobutton (Abb. 4.5) oder aus einer Liste ausgewählt werden.

- **Metamodelltypen**
  Im Unterschied zu den Grundtypen (`String`, `Int` und `Boolean`) können Instanzen von Metamodelltypen, d.h. Modellelemente, nicht in einem Textfeld eingegeben werden, da die Namen von Modellelementen im Allgemeinen nicht eindeutig sind: ein gültiges Modell darf beispielsweise zwei Komponenten mit dem Namen `"Slot"` enthalten. Die einzige Eingabemöglichkeit besteht damit in der Auswahl des gewünschten Elements aus den vorhandenen Modellelementen. Die Auswahl kann zurzeit in einer Liste oder einer Gruppe von Radiobuttons stattfinden.

  Die Abbildung 4.6 zeigt den Eingabedialog für die Variable `comp2`, der bei der Auswertung der ODL-Abfrage
  `context comp1:Component. context comp2:Component. comp1=comp2`
  gestartet wird. Eine Liste im Dialog führt alle im Modell verfügbaren Komponenten auf, von denen eine ausgewählt werden muss. Die Auswahl von Modellelementen anderer Metamodelltypen (z.B. `Port`, `Channel` usw.) gestaltet sich analog.

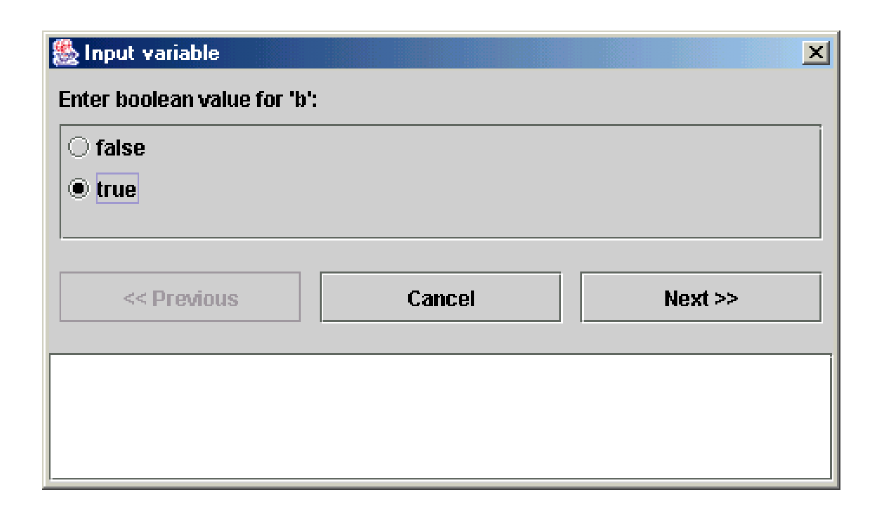


Abbildung 4.5: Eingabe eines `Boolean`-Werts über Radiobuttons

4.2.3 Eingabe von Produktwerten

Für die Eingabe eines Produktwerts werden im Eingabedialog die Eingabepanels für alle Elemente des entsprechenden Produkttyps nebeneinander angezeigt. Betrachten wir als Beispiel die Abfrage

```
context var:(c1:Component,c2:Component,b:Boolean).true          (4.9)
```

Der von dieser Abfrage geöffnete Eingabedialog (Abb. 4.7) enthält im Eingabebereich je ein Eingabepanel für jedes Element des Produkttyps. Da ein Produkttyp beliebig viele Elemente enthalten darf, kann auch der Eingabebereich bei einem Produkttyp beliebig breit werden. Deshalb wird der Eingabebereich mit Scrollbalken und mit einem Zoomslider ausgestatten, über den der Maßstab für die Breite des Eingabebereichs festgelegt werden kann. Wird der Zoomslider nicht mehr benötigt, kann er versteckt werden (Abb. 4.8): dafür muss der Benutzer den rechten Button der Splittinglinie zwischen dem Zoomslider und dem Eingabebereich betätigen.

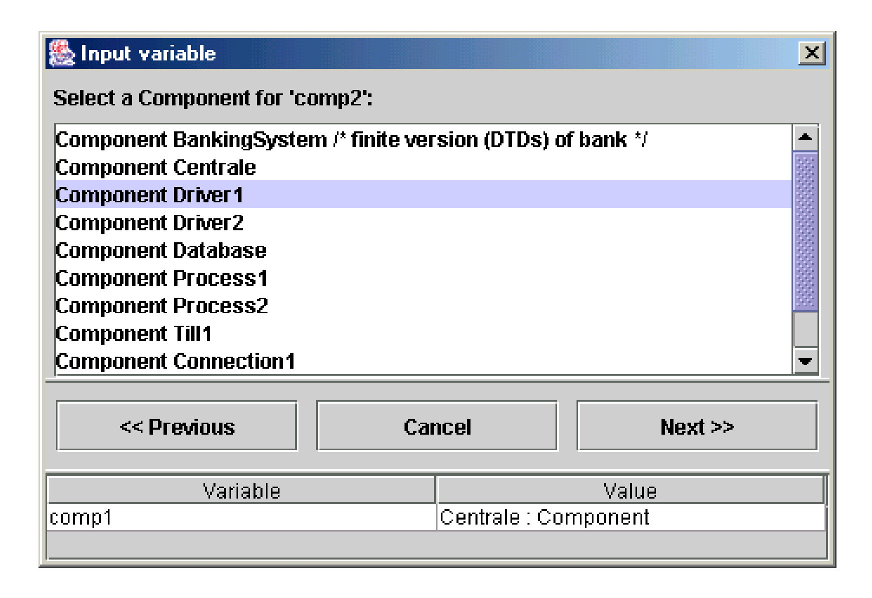


Abbildung 4.6: Auswahl einer Komponente

Die Eingabe eines Produktwerts kann abgeschlossen werden, wenn für alle Produkttyp-Elemente ein zulässiger Wert eingegeben wurde.

4.2.4 Eingabe von eingeschränkten Typen

Ein eingeschränkter Typ unterscheidet sich von seinem Basistyp nur durch die angewendete Restriktionsbedingung. Deshalb gleicht der Eingabedialog für einen eingeschränkten Typ dem Eingabedialog für seinen Basistyp. So würde der Eingabedialog für die Abfrage

```
context var:{ v:(c1:Component,c2:Component,b:Boolean) |
is SubComponents( v.c1, v.c2 ) } .true
```
(4.10)

genauso aussehen, wie der Eingabedialog für die Abfrage 4.9 auf der Abbildung 4.7 – nur die Überschrift im Eingabebereich würde `”Restricted type variable ’var’”` anstatt `”Product type variable ’var’”` lauten. Der wichtige Unterschied in der Funktionsweise ist, dass die Eingabe nur dann abgeschlossen werden kann, wenn der eingegebene Wert des Basistyps die Restriktionsbedingung erfüllt. Bei der Abfrage 4.10 würde der *Next*-Button des Eingabedialog nur dann aktiviert, wenn `c2` eine Unterkomponente von `c1` ist.

4.2.5 Eingabe von Mengen

Der Eingabebereich des Eingabedialogs für Mengen ist komplexer als alle anderen Eingabebereiche und besteht aus zwei Teilen. Betrachten wir den Eingabedialog für die ODL-Abfrage

```
context stringSet:set String.true
```

auf der Abbildung 4.9:

- Basistyp-Eingabebereich:
  im unteren Teil des Mengen-Eingabepanels befindet sich ein Eingabepanel für die Werte des Basistyps der Menge (in unserem Beispiel ist es `String`) sowie die Buttons *Add* und *Clear*.

  Der *Add*-Button fügt einen eingegebenen Basistyp-Wert zur Menge hinzu, falls er zulässig ist und nicht bereits in der Menge enthalten ist. Die Bestätigung des Basistyp-Werts durch das Drücken der Eingabe-Taste has dieselbe Funktion. Der *Clear*-Button löscht den aktuellen Wert im Basistyp-Eingabepanel.

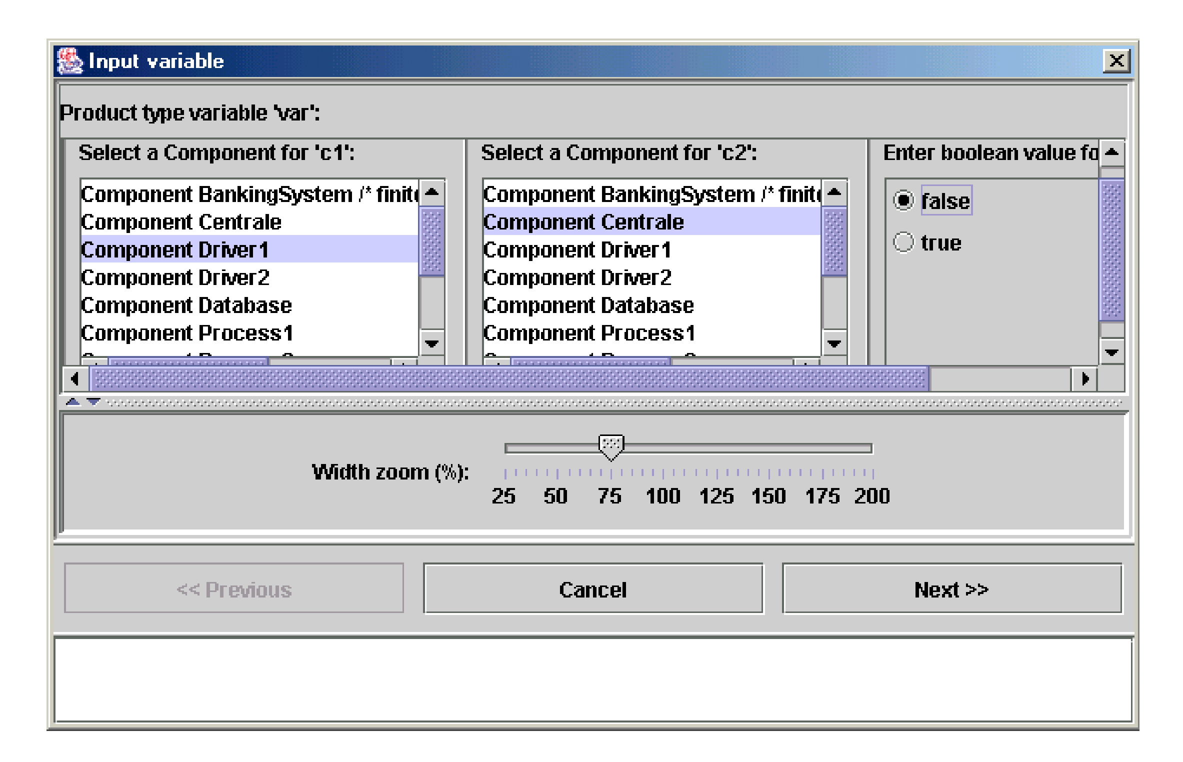


Abbildung 4.7: Eingabe eines Produkttyps

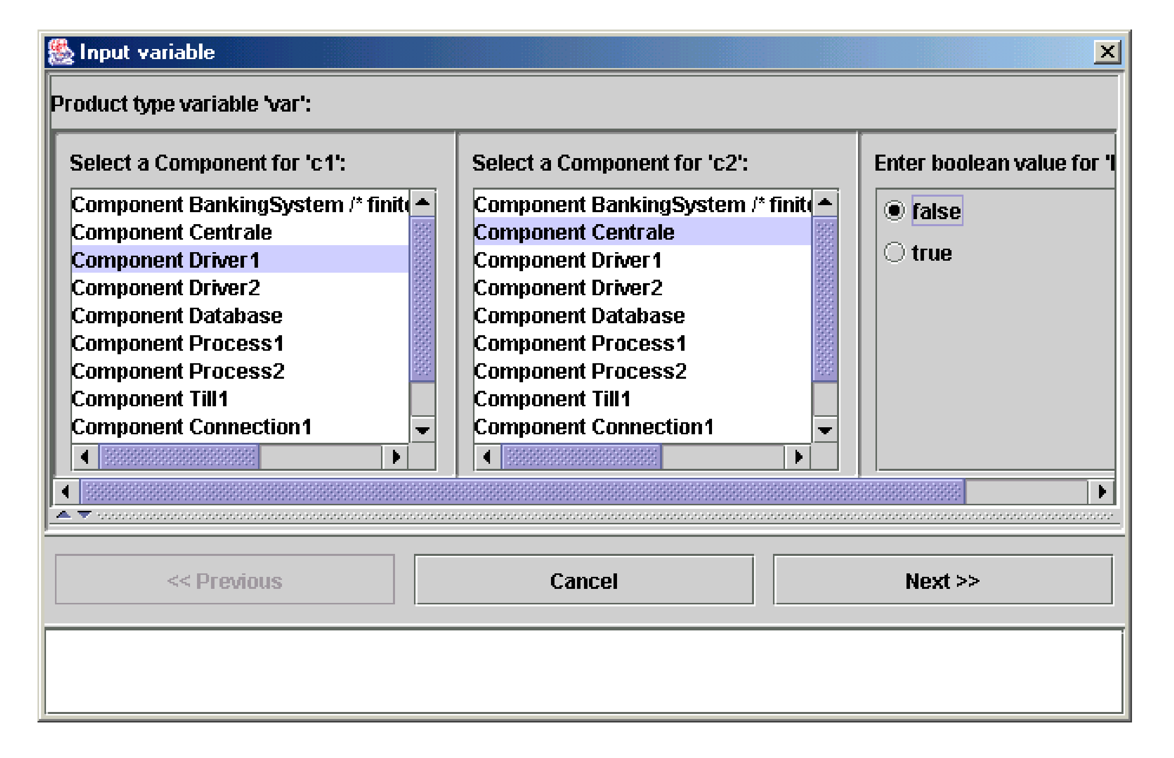


Abbildung 4.8: Eingabe eines Produkttyps, Zoomslider minimiert

- Mengenelemente:
  der obere Teil des Mengen-Eingabepanels zeigt die bisher eingegebenen Mengenelemente und den Button *Remove*, mit dem Mengenelemente aus der Menge wieder entfernt werden können.

  Neben der im Dialog auf der Abbildung 4.9 verwendeten Anzeige der Mengenelemente in einer Liste kann für die Anzeige der Mengenelemente auch eine Tabelle benutzt werden (Abb. 4.10). Letztere ist insbesondere für die Eingabe von Produkttyp-Mengen von Vorteil, da jedes Produkttyp-Element in einer separaten Tabellenspalte angezeigt wird.

Die beiden Teile des Mengen-Eingabebereichs werden durch eine Splittinglinie getrennt, mit deren Hilfe das Größenverhältnis zwischen den beiden Teilen jederzeit verändert werden kann oder auch einer der Teile zeitweise versteckt werden kann, um eine bessere Übersicht über den jeweils anderen Teil zu bekommen.

4.2.6 Eingabe von eingeführten Typen

Als *eingeführte Typen* bezeichnen wir Ausdrücke, die zwar keinen echten Typ darstellen, wohl aber – wie jeder echte Typ – eine Iteration über ihre Instanzen zulassen und in allen ODL-Sprachkonstrukten außer dem `new`-Quantor mit dem vorangestellten Schlüsselwort `element` wie jeder andere Typ verwendet werden können. Zurzeit agieren Mengenvariablen und Ausdrücke, die mengenwertige Ergebnisse zurückliefern, als eingeführte Typen (s. auch S. 23).

Das Eingabepanel, welches Eingabedialoge für eingeführte Typen verwenden, ähnelt denen für Metamodelltypen: es handelt sich um ein Auswahlpanel (als Liste oder Radiobuttons), in dem ein Element des eingeführten Typs auszuwählen ist. Betrachten wir folgende Beispiele für die Eingabe des Werts eines eingeführten Typs:

- Mengenvariable:
  In der ODL-Abfrage
  ```
  context intSet:set Int. context i:element intSet. true
  ```
  muss zunächst eine Menge von ganzen Zahlen eingegeben werden, und danach ist eine der Zahlen aus der Menge `intSet` für die Variable `i` auszuwählen. Die Abbildung 4.11 zeigt den für die Variable `i` geöffneten Eingabedialog, nachdem für `intSet` zuvor die Zahlen `{1,3,5,8,12,15}` eingegeben wurden.

- Ausdruck mit mengenwertigem Ergebnis:
  Als Beispiel für die Auswahl eines Elements aus dem mengenwertigen Ergebnis eines Ausdrucks betrachtet wir folgende ODL-Abfrage:
  ```
  context c:Component. context channel:element(c.Channels).true
  ```
  hier wird zunächst eine Komponente ausgewählt und anschließend muss ein Kanal aus der Liste aller zur Komponente gehörenden Kanäle ausgewählt werden (Abb. 4.12).

Generell ist die Auswahl eines Elements aus einer Menge, falls möglich, immer der Verwendung eines entsprechenden eingeschränkten Typs vorzuziehen, denn eine Abfrage der Form
```
context var:element( entity.SomeAssociation ).true
```
ist stets effizienter und bequemer als eine Abfrage der Form
```
context var:{v:type | is SomeAssociation( entity, v )}.true
```
.

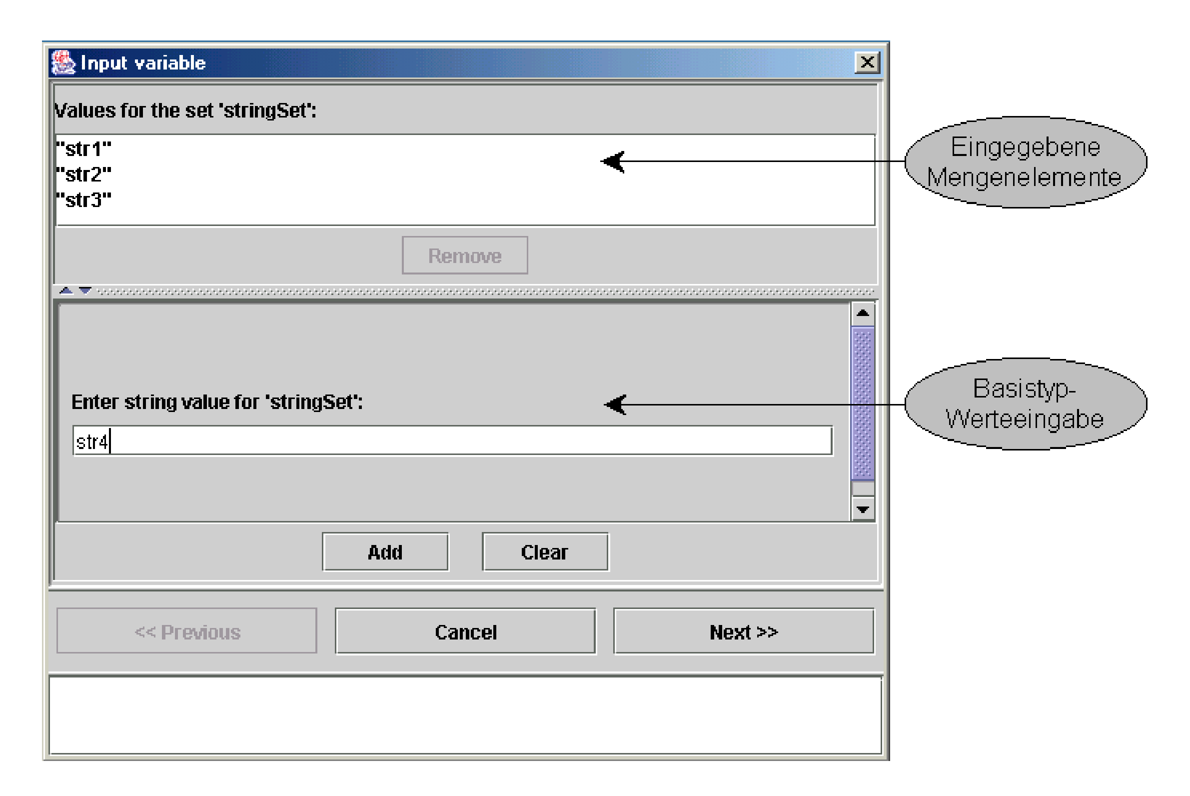


Abbildung 4.9: Eingabe einer Menge von Strings

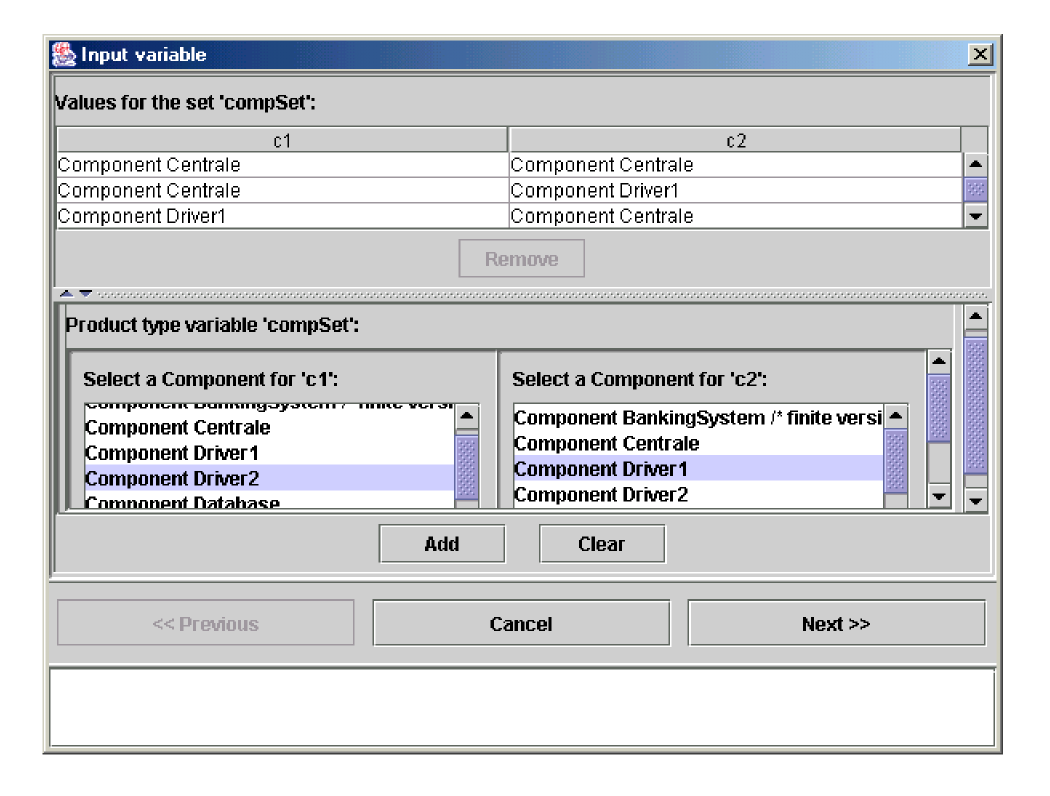


Abbildung 4.10: Eingabe einer Menge von Produktwerten

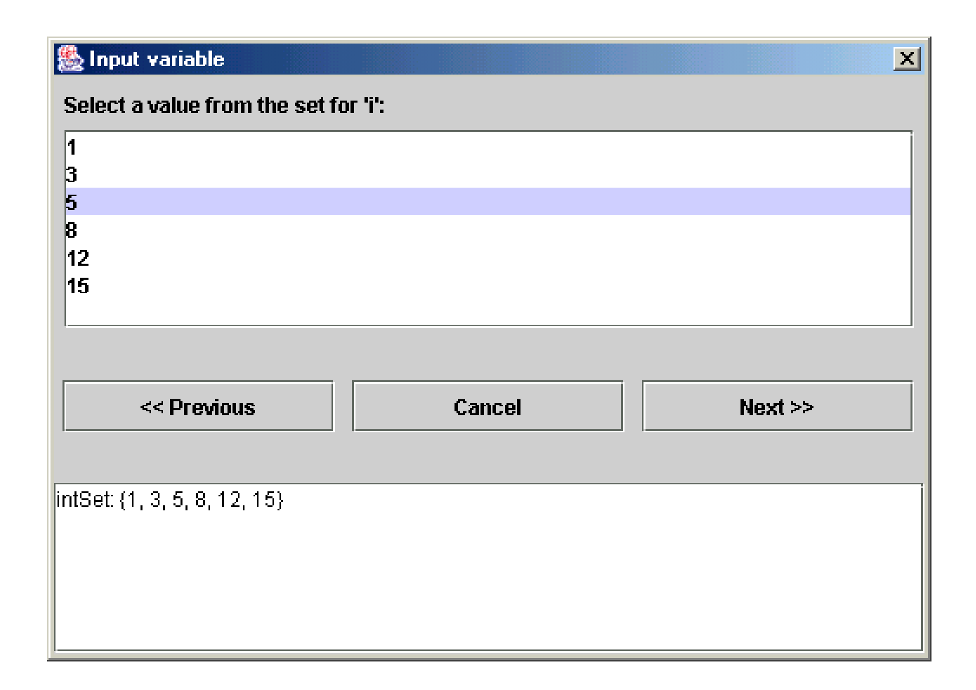


Abbildung 4.11: Auswahl einer Zahl aus einer zuvor eingegebenen Menge

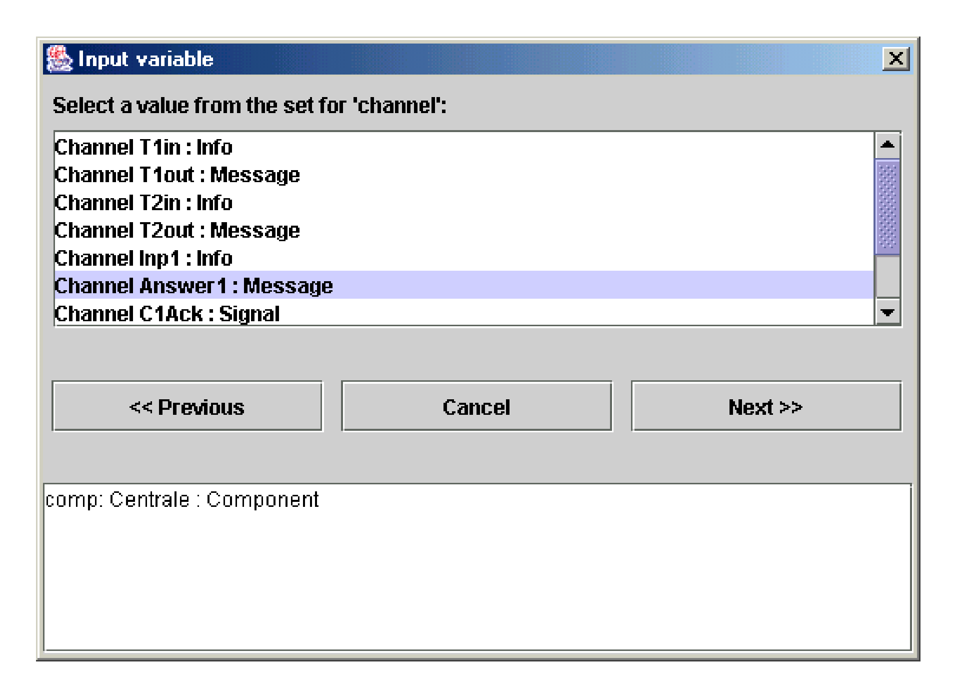


Abbildung 4.12: Auswahl eines Kanals aus der Menge der Kanäle einer zuvor ausgewählten Komponente

4.2.7 Konfiguration der Eingabeschnittstelle

Wie bereits früher erwähnt, können für die Eingabedialoge für verschiedene Datentypen diverse Einstellungen vorgenommen werden, darunter der Typ des zu verwendenden Dialogfenster, die Art der Werteanzeige usw.

Beim Öffnen des Konfigurationsdialogs werden die aktuellen Einstellungen angezeigt. Nach der Änderung der Einstellungen können sie entweder mit dem *OK*-Button übernommen oder mit dem *Cancel*-Button verworfen werden. Im Folgenden erläutern wir die möglichen Einstellungen für alle Datentypen.

Der Konfigurationsdialog wird im dem ODL-Editor über den neuen Menüpunkt `Options` → `Query Options` aufgerufen (Abb. 4.13).

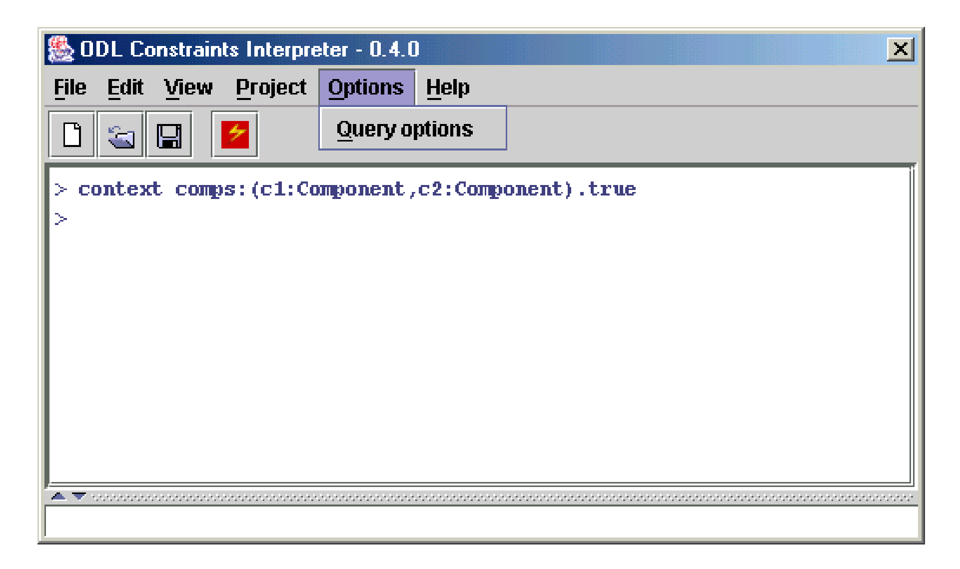


Abbildung 4.13: ODL-Editor, Aufruf des Konfigurationsdialogs

Für jeden der Datentypen `Boolean`, `Int`, `String`, `Entity`, `IntroducedType`, `ProductType` und `SetType` enthält der Konfigurationsdialog ein Konfigurationspanel, in dem die Einstellungen für die Eingabe von Werten des jeweiligen Typs eingesehen und verändert werden können.

Die Konfigurationspanels sind in einem hohen Fenster untergebracht, das über eine vertikale Bildlaufleiste gescrollt werden kann. Die Konfigurationspanels sind für alle Datentypen bis auf `SetType` in gleicher Weise aufgebaut und bestehen aus folgenden Teilbereichen (Abb. 4.14):

- Bereich **Value input**:
  Auswahl des Eingabepanels, in dem Werte des betreffenden Datentyps eingegeben werden sollen.

- Bereich **Display known variable in**:
  Auswahl des Werteanzeigebereichs, in dem bereits bekannte Variablen angezeigt werden.

- Bereich **Query dialog**:
  Typ des Dialogfensters, das als Eingabedialog für den betreffenden Datentyp verwendet werden soll.

- Checkbox **Use shared dialog instance**:
  Über diese Checkbox wird festgelegt, ob gemeinsam mit anderen Datentypen dieselbe Eingabedialog-Instanz verwendet werden soll, oder eine eigene Dialoginstanz für diesen Datentyp erstellt und verwendet werden soll. Die Auswirkung dieser Einstellung ist die folgende: wird für mehrere Datentypen derselbe Eingabedialogtyp eingestellt und die Checkbox **”Use shared dialog instance”** gesetzt, so bilden sie eine Gruppe, für die eine gemeinsame Dialoginstanz

verwendet wird. Dialogfenster-Einstellungen wie Fensterposition und -größe werden von allen Datentypen aus dieser Gruppe geteilt – Einstellungen, die während der Eingabe eines Wert für einen der Datentypen aus der Gruppe vorgenommen wurden, werden automatisch für alle Datentypen aus dieser Gruppe wirksam. Wird für einen Datentyp hingegen eine eigenständige Dialoginstanz verwendet (Checkbox nicht gesetzt), so sind auch alle fensterspezifischen Einstellungen des Eingabedialogs für diesen Datentyp individuell.

Die Abbildung 4.14 zeigt einen Ausschnitt des Konfigurationsdialogs mit den Konfigurationspanels für die Datentypen `Boolean`, `Int` und `String`. Als Eingabepanel (Bereich **Value input**) kann eines der aufgelisteten Panels ausgewählt werden, wobei die Auswahl je nach Datentyp variieren kann. Als Werteanzeigekomponente (Bereich **Display known variables in**) wird stets ein Textbereich und eine Tabelle angeboten. Als Dialogfester für die Eingabe (Bereich **Query dialog**) kann ein einfaches Dialogfenster (Abb. 4.1) oder ein Dialogfenster mit veränderbaren Bereichsgrößen (Abb. 4.4) ausgewählt werden. Die Einstellung **Use shared dialog instance** für den auf auf der Abbildung 4.14 gezeigten Fall bewirkt (zusammen mit der Auswahl des gleichen Dialogfenstertyps für alle Datentypen), dass für die Datentypen `Boolean`, `Int` und `String` dieselbe Eingabedialog-Instanz benutzt wird.

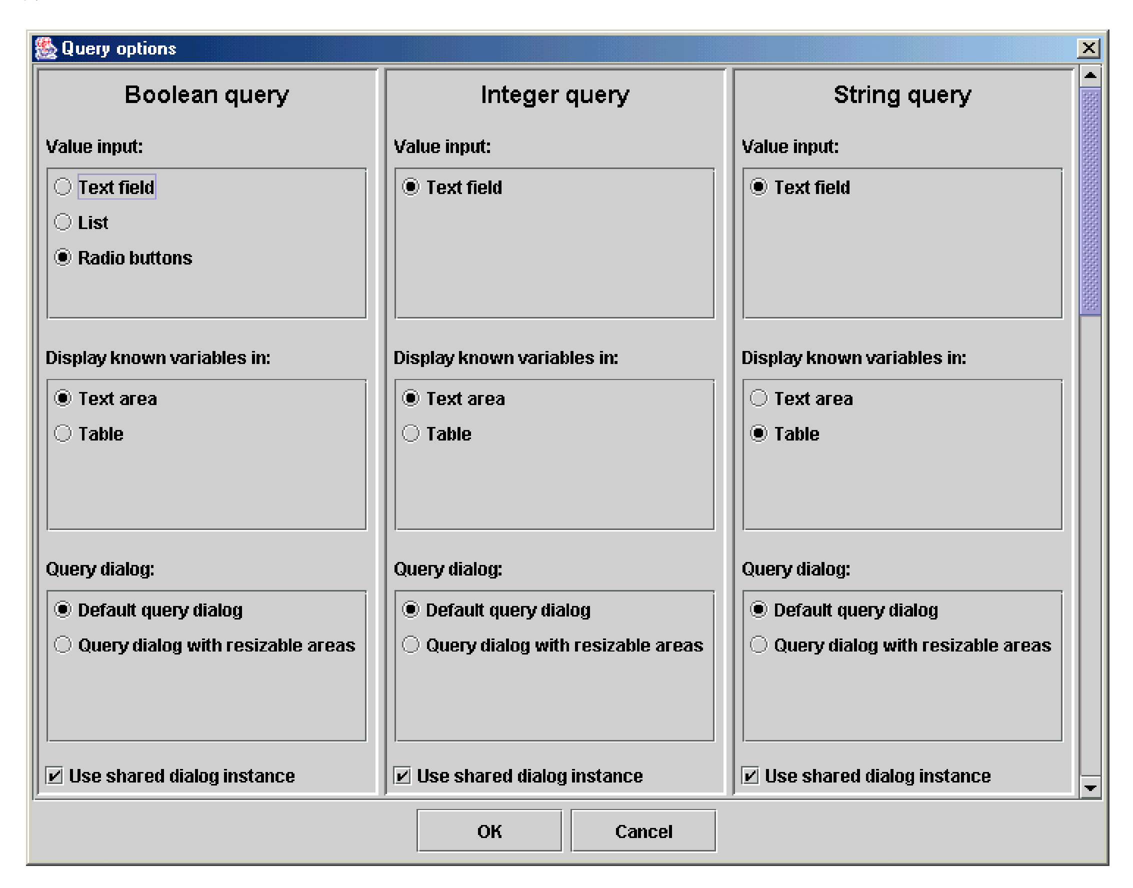


Abbildung 4.14: Konfiguration der Eingabedialoge für `Boolean`, `String` und `Int`

Das Konfigurationspanel für die Eingabe von `Entity`-Werten gleicht dem Konfigurationspanel für `IntroducedType` und wird daher nicht gesondert besprochen.

Auf der Abbildung 4.15 werden die Konfigurationspanels für die Eingabe von Datentypen `IntroducedType` und `ProductType` gezeigt. Sie unterscheiden sich von den Konfigurationspanels auf der Abbildung 4.14 nur in der angebotenen Auswahl der Eingabepanels. Für `IntroducedType` stehen als Eingabepanels eine Liste und Radiobuttons zur Verfügung. Für die Eingabe von `ProductType` steht nur ein Eingabepanel zur Verfügung, das für einen einzugebenden `ProductType`-Wert die Eingabepanels für die einzelnen Produkttyp-Elemente nebeneinander anzeigt

(s. Abschnitt 4.2.3 und Abbildung 4.7).

Das Konfigurationspanel für die Eingabe von SetType-Werten auf der Abbildung 4.16 zeigt einen zusätzlichen Konfigurationsbereich **Variable value**, in dem festgelegt wird, wie die in der eingegebenen Mengenvariable enthaltenen Basistyp-Werte angezeigt werden sollen. Dafür kann eine Liste oder eine Tabelle verwendet werden. Wie bereits für ProductType steht für die Eingabe von SetType-Werten nur ein Eingabepanel zur Verfügung, dass speziell für Mengen entwickelt wurde (s. Abschnitt 4.2.5 und Abbildung 4.9).

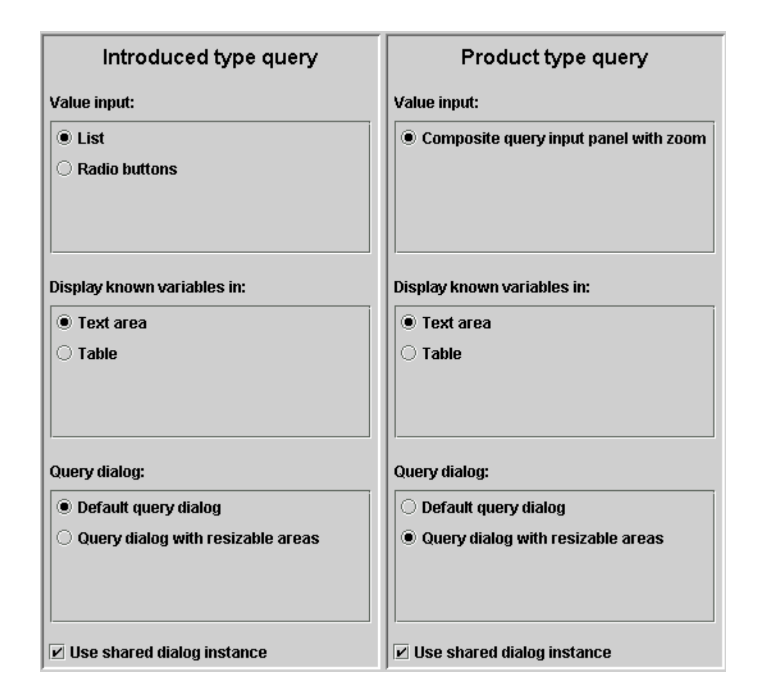


Abbildung 4.15: Konfiguration der Eingabedialoge für IntroducedType und ProductType

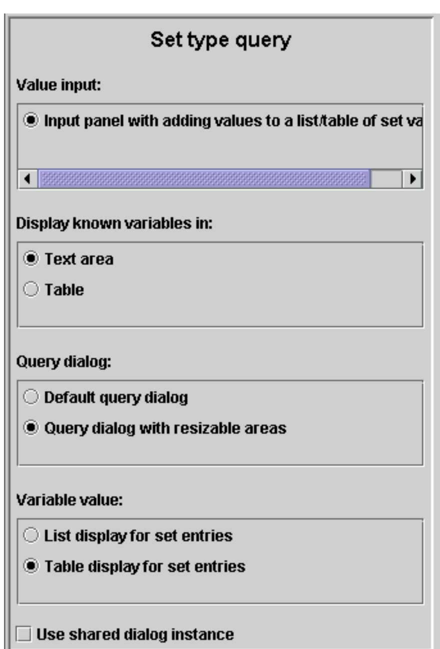


Abbildung 4.16: Konfiguration des Eingabedialogs für SetType

Für die Eingabe von RestrictedType-Werten ist keine Eingabedialog-Konfiguration notwendig, weil der Eingabedialog für einen RestrictedType äußerlich dem Eingabedialog für seinen Basistyp gleicht (s. Abschnitt 4.2.4) – es werden somit auch die Eingabedialog-Einstellungen des Basistyps verwendet.

4.3 Beispiele von ODL-Abfragen

Am Schluss dieses Kapitels möchten wir noch einige praktische Beispiele für ODL-Abfragen geben.

- ```
  exists c:Component. is Name( c, ”comp1” )
  ```
  Alle Komponenten mit dem Namen ”comp1” finden.

- ```
  exists c1:Component.exists c2:Component.(
    is Name( c1, ”comp1” ) and is SubComponents( c1, c2 ))
  ```
  Alle Unterkomponenten der Komponente ”comp1” finden.

- ```
  exists comps:(c1:Component, c2:Component).(
    is Name( comps.c1, ”comp1” ) and
    is SubComponents( comps.c1, comps.c2 ))
  ```
  Umformulierung der vorherigen Abfrage mit Verwendung eines Produkttyps – sie wird dadurch schneller ausgewertet wird.

- ```
  context comps:{ c:(c1:Component, c2:Component) |
    is SubComponents( c.c1, c.c2 ) }. true
  ```
  Eine Komponente und eine ihrer Unterkomponenten auswählen.

- ```
  exists ports:( p1:Port, p2:Port ).(
    neg ports.p1 = ports.p2 and
    ports.p1.Name = ports.p2.Name )
  ```
  Alle Portpaare verschiedener Ports mit gleichem Namen finden.

- ```
  exists port:Port. port.Type.Text = ”Int”
  ```
  Alle Ports mit dem Datentyp `Int` finden.

- ```
  exists ports:( p1:Port, p2:Port ).(
    neg ports.p1 = ports.p2 and
    ports.p1.Type.Text = ports.p2.Type.Text )
  ```
  Alle Portpaare verschiedener Ports mit gleichem Datentyp finden.

- ```
  context c1:Component.context name:String.new c2:Component.(
    result has Name( c2, name ) and
    result has SubComponents( c1, c2 ))
  ```
  Eine Komponente und den Namen für eine neue Unterkomponente eingeben: es wird eine neue Komponente erstellt, die den eingegebenen Namen erhält und als Unterkomponente der ausgewählten Komponente ins Modell eingefügt wird.

- ```
  context comp:Component. context names:set String.
    forall name:element names.( new c:Component.(
      result has Name( c, name ) and
      result has SubComponents( comp, c )))
  ```
  Eine Komponente auswählen und anschließend eine Menge von Namen eingeben: für jeden Namen aus der eingegebenen Menge wird bei der ausgewählten Komponente eine Unterkomponente mit diesem Namen erstellt.

- ```
  context portSet:set Port.
    forall port:element portSet.(
      context name:String. result has Name( port, name ) )
  ```
  Eine Portmenge auswählen und anschließend für jeden Port aus der Menge einen neuen Namen spezifizieren.

- ```
  context comp:Component.
    result not has SubComponents( comp.SuperComponent, comp )
  ```
  Entfernt die ausgewählte Komponente aus dem Modell, indem sie aus der Liste der Unterkomponenten seiner Oberkomponente entfernt wird.

- ```
  portsBelongToChannel( ch:Channel, p1:Port, p2:Port ) :=
    (is SourcePort( ch, p1 ) and is DestinationPort( ch, p2 )) or
    (is SourcePort( ch, p2 ) and is DestinationPort( ch, p1 ))
  ```
  Dieses benamte Prädikat überprüft, ob die als Parameter übergebenen Ports gerade die Eingangs- und Ausgangsports des übergebenen Kanals sind.

- ```
  componentsConnected( c1:Component, c2:Component ) :=
    exists ports:( p1:element(c1.Ports), p2:element(c2.Ports) ).(
      ( exists ch1:element (ports.p1.OutChannels).
        ch1 = ports.p2.InChannel ) or
      ( exists ch2:element (ports.p2.OutChannels).
        ch2 = ports.p1.InChannel ) )
  ```
  Das benamte Prädikat `componentsConnected` stellt für zwei Komponenten fest, ob sie durch mindestens einen Kanal verbunden sind.

- ```
  forall c:Component.(
    ( exists c2:Component. is SubComponents( c, c2 ) ) or
    ( exists a:Automaton. is Automaton( c, a ) ) )
  ```
  Mithilfe dieser Abfrage wird für ein Modell die Konsistenzbedingung überprüft, dass jede Komponente eine Verfeinerung in Form von Unterkomponenten oder eines das Verhalten der Komponente bestimmenden Automaten besitzt.

- ```
  context c:Component. forall p:element( c.Ports ).
    new locVar:LocVariable.(
      result has Name( locVar, p.Name ) and
      result has Type( locVar, p.Type ) and
      result has LocVariables( c, locVar ) )
  ```
  In einer vom Benutzer ausgewählten Komponenten wird für jeden Port eine lokale Variable erstellt, die denselben Typ und Namen wie der Port hat. Diese Variablen können dann beispielsweise als Zwischenspeicher für die von Ports empfangenen bzw. versendeten Daten benutzt werden.

- ```
  context comps:( dest:Component,
                  src:src:Component|size(src.SubComponents)>0 ).(
    forall subComp:element( comps.src.SubComponents ).(
      result not has SubComponents( comps.src, subComp ) and
      result has SubComponents( comps.dest, subComp ) ) and
    forall channel:element( comps.src.Channels ).(
      result not has Channels( comps.src, channel ) and
      result has Channels( comps.dest, channel ) ) and
    forall port:element( comps.src.Ports ).(
      result not has Ports( comps.src, port ) and
      result has Ports( comps.dest, port ) ) and
    forall locVar:element( comps.src.LocVariables ).(
      result not has LocVariables( comps.src, locVar ) and
      result has LocVariables( comps.dest, locVar ) ) )
  ```
  Diese Abfrage dient zur Zusammenführung zweier Komponenten: nachdem der Benutzer eine

Quellkomponente und eine Zielkomponente auswählt, werden alle Unterkomponenten, Kanäle, Ports und lokale Variablen der Zielkomponente in die Quellkomponente verschoben.

Diese Formulierung ist nur auf Quellkomponenten anwendbar, die Unterkomponenten und keinen Automaten besitzen (nach den Regeln des QUEST-Metamodells, darf eine Komponente entweder Unterkomponenten zur Beschreibung der Struktur oder einen Automaten zur Beschreibung des Verhaltens besitzen).

Nun wollen wir noch ein komplexeres Beispiel präsentieren. Die folgende ODL-Abfrage erstellt eine neue Zwischenkomponente und fügt sie in das vom Benutzer ausgewählte Kanalbündel ein, wobei jedes Kanal aus dem Bündel durch zwei neue Kanäle ersetzt wird, welche die Ports des ersetzten Kanals mit neuen Ports der eingefügten Komponente verbindet:

```
/* Hauptkomponente auswählen */
context mainComponent:Component.
  /* Kanalbündel aus der Hauptkomponenten auswählen */
  context channelBundle:set element(mainComponent.Channels).
  /* Den Namen für die neue Zwischenkomponente eingeben */
  context midCompName:String.
  /* Zwischenkomponente in der Hauptkomponente erstellen */
  new midComp:Component.(
  result has Name( midComp, midCompName) and
  result has SubComponents( mainComponent, midComp ) and
  /* Alle Kanäle aus dem ausgewählten Kanalbündel auftrennen und
  durch die Zwischenkomponente "durchleiten" */
  forall channel:element channelBundle.(
    /* Eingangs- und Ausgangsport des Kanals merken */
    exists chPorts:(
    in:{ p:element( channel.DestinationPort.Component.Ports ) |
      is DestinationPort( channel, p )},
    out:{ p:element( channel.SourcePort.Component.Ports ) |
      is SourcePort( channel, p ) } ).(
    /* Kanal löschen */
    result not has Channels( mainComponent, channel ) and
    result not has OutChannels( chPorts.out, channel ) and
    result not has InChannel( chPorts.in, channel ) and
    /* Eingangsport in der Zwischekomponente erstellen */
    new inPort:Port.(
    result has Name( inPort, channel.Name ) and
    result has Type( inPort, channel.Type ) and
    result has Direction( inPort, chPorts.in.Direction ) and
    result has Ports( midComp, inPort ) and
    /* Kanal zwischen dem Ausgangsport des gelöschten Kanals und
    dem neuen Eingangsport erstellen */
    new inChannel:Channel.(
    result has Name( inChannel, channel.SourcePort.Name ) and
    result has Type( inChannel, channel.Type ) and
    result has SourcePort( inChannel, chPorts.out ) and
    result has DestinationPort( inChannel, inPort ) and
    result has OutChannels( chPorts.out, inChannel ) and
    result has InChannel( inPort, inChannel ) and
    result has Channels( mainComponent, inChannel ))) and
    /* Ausgangsport in der Zwischekomponente erstellen */
```

```
    new outPort:Port.(
    result has Name( outPort, channel.Name ) and
    result has Type( outPort, channel.Type ) and
    result has Direction( outPort, chPorts.out.Direction ) and
    result has Ports( midComp, outPort ) and
    /* Kanal zwischen dem neuen Ausgangsport und
    dem Eingangsport des gelöschten Kanals erstellen */
    new outChannel:Channel.(
    result has Name( outChannel, channel.DestinationPort.Name ) and
    result has Type( outChannel, channel.Type ) and
    result has SourcePort( outChannel, outPort ) and
    result has DestinationPort( outChannel, chPorts.in ) and
    result has OutChannels( outPort, outChannel ) and
    result has InChannel( chPorts.in, outChannel ) and
    result has Channels( mainComponent, outChannel )))
  )))
```

Abschließend wollen wir noch einige Beispiele vorstellen, die mit der Verwendung des Fixpunktoperators möglich werden, der kurz nach der Fertigstellung des Implementierungsteils der vorliegenden Arbeit von einem anderen Entwickler realisiert wurde.

- Rekursives Finden aller Komponenten, die im Strukturbaum einer vom Benutzer ausgewählten Komponente vorkommen, d.h., für die ausgewählten Komponente werden alle Unterkomponenten, dann alle Unterkomponenten der Unterkomponenten usw. in einer Menge gesammelt:

```
context mainComp:Component.
  exists subStructure:lfp subComps set c:Component with (
    c = mainComp or exists superComp:element subComps.
      is SubComponents( superComp, c )
  ).true
```

- Konsistentes Umbenennen einer Kette von Kanälen: gibt es zwischen Komponenten $c_0, c_1, \ldots,$ c_n Kanäle $ch_1, \ldots, ch_n$, die alle denselben Namen haben, so können alle Kanäle aus dieser Kette umbenannt werden, indem der Benutzer einen Kanal c_i auswählt, den neuen Namen eingibt, und anschließend die anderen Kanäle aus der Kette mithilfe des Fixpunktoperators gefunden und umbenannt werden:

```
context p:( channel:Channel, newName:String ).
 exists channelSet:lfp chSet set ch:Channel with (
  ch = p.channel or exists ch2:element chSet.(
   ch2.Name = ch.Name and (
    /*Check for all input ports of channel's source
      component, whether they are connected to the channel*/
    (exists pIn:element(ch2.SourcePort.Component.Ports).
      ch.DestinationPort = pIn ) or
    /*Check for all output ports of channel's destination
      component, whether they are connected to the channel*/
    (exists pOut:element(ch2.DestinationPort.Component.Ports).
      ch.SourcePort = pOut ) ) ) ).
forall channel: element channelSet.
 result has Name( channel, p.newName )
```

Kapitel 5

Implementierung

In diesem Kapitel wird technische Implementierung der Erweiterungen des ODL-Systems beschrieben, die im Kapitel 4 vorgestellt wurden. Wir beschreiben die Implementierung der Erweiterung des ODL-Sprachumfangs im Abschnitt 5.1 und die Realisierung der interaktiven Schnittstelle für Benutzerabfragen im Abschnitt 5.2.

Zunächst wollen wir auf den allgemeinen Aufbau des ODL-Systems eingehen, der schematisch auf der Abbildung 5.1 gezeigt wird (eine ausführliche Beschreibung der ODL-Systemarchitektur findet sich in [Pasch], S.15-40).

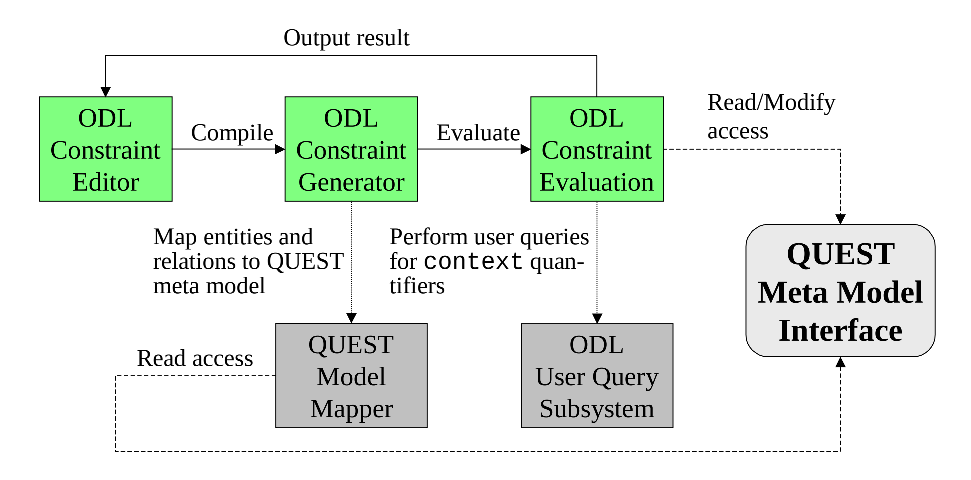


Abbildung 5.1: Schematische Darstellung des ODL-Systems

Die Komponenten `"ODL Costraint Editor"`, `"ODL Costraint Generator"` und `"ODL Costraint Evaluator"` bilden den Hauptkreislauf des ODL-Systems – eine ODL-Abfrage wird im Editor erstellt, dann wird sie vom Generator zu einer internen Darstellung kompiliert, danach ausgewertet, und schließlich wird das Ergebnis im ODL-Editor ausgegeben.

Der `"ODL Costraint Generator"` greift bei Bedarf auf den `"QUEST Model Mapper"` zu, um Entitäten und Relationen in ODL auf Metamodell-Elemente sowie ihre Attribute und Assoziationen im QUEST-Metamodell abzubilden. Bei der Auswertung einer ODL-Abfrage muss der `"ODL Costraint Evaluator"` für die Auswertung von `context`-Quantoren auf das `"ODL User Query Subsystem"` zugreifen, damit der Benutzer einen Wert für die vom `context`-Quantor gebundene Variable eingeben kann.

Das `"QUEST Meta Model Interface"` stellt die Schnittstelle zum QUEST-Metamodell dar, über die das ODL-System auf das Metamodell und auf Produktmodelle zugreifen kann. Diese Schnittstelle, die eine *Fassade*-Implementierung ([GammaEtAl], S.212-222) darstellt, ist vom Grundsatz her nicht an eine konkrete Metamodell-Implementierung gebunden – ihre Zugriffsmechanismen können an neue Metamodell-Implementierungen angepasst werden, ohne dass die für das ODL-

System sichtbare Schnittstelle sich wesentlich ändert, sodass das ODL-System für die Arbeit mit einem neuen Metamodell nicht oder geringfügig modifiziert werden muss.

Die Tabelle 5.1 führt die im Rahmen der vorliegenden Arbeit realisierten Änderungen an verschiedenen Komponenten des ODL-Systems auf.

Komponenten	Änderungen
`ODL Constraint Generator`	Der ODL-Abfragegenerator wurde an die Erweiterung bestehender und Einführung neuer Sprachkonstrukte angepasst. Die vom Generator benutzte SableCC-Grammatik für ODL wurde erweitert.
`ODL Constraint Evaluation`	Auswertungsklassen für neue Sprachkonstrukte wurden hinzugefügt, vorhandene Auswertungsklassen wurden an Erweiterungen der ihnen entsprechenden Sprachkonstrukte angepasst.
`ODL User Query Subsystem`	Eine interaktive Benutzerschnittstelle für die Eingabe von Variablenwerten wurde entwickelt und in das User-Query-Subsystem integriert (die frühere provisorische Implementierung lieferte leere Ergebnisse als Benutzereingabe zurück).
`ODL Constraint Editor`	Geringfügige Anpassungen und Optimierungen wurden vorgenommen.

Tabelle 5.1: Änderungen an Komponenten des ODL-Systems

5.1 Erweiterung des Sprachumfangs

Bevor wir die Implementierung der Erweiterungen des Sprachumfangs im Detail beschreiben, wollen wir einen kurzen Überblick über die Implementierung des ODL-Auswertungs-Subsystems geben (eine ausführliche Beschreibung findet sich in [Pasch], S.15-40).

Die Abbildung 5.2 stellt noch einmal die Verarbeitung einer ODL-Abfrage von der Eingabe bis zur Berechnung des Ergebnisses detaillierter dar.

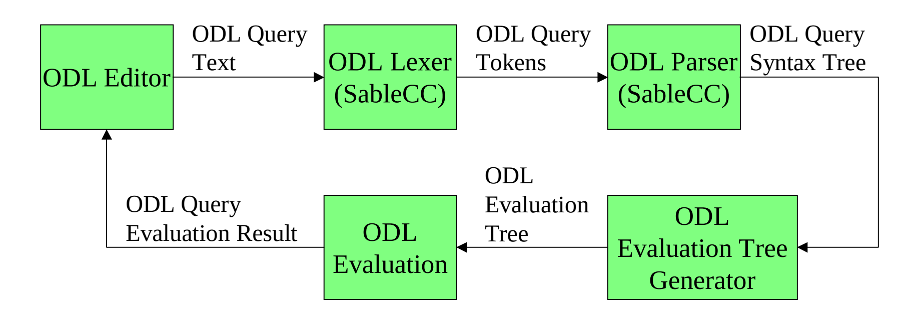


Abbildung 5.2: Verarbeitung einer ODL-Abfrage

Eine ODL-Abfrage, die vom Benutzer im ODL-Editor erstellt wird, muss für die Auswertung zunächst von einem Lexer in Tokens zerlegt und anschließend von einem Parser analysiert werden, der einen Syntaxbaum aufbaut. Dafür wird der von SableCC aus der ODL-Grammatik (s. Anhang C) automatisch generierte Lexer und Parser verwendet – sie befinden sich in den Packages `quest.odl.parser.lexer` bzw. `quest.odl.parser.parser`. Der Parser liefert als Ergebnis einen Syntaxbaum, in dem jedem Token aus der ODL-Abfrage ein Knoten entspricht. Alle möglichen Syntaxbaum-Knoten, die ebenfalls automatisch aus der ODL-Grammatik generiert werden, befinden sich im Package `quest.odl.parser.node` und erben von der abstrakten Klasse `Node`. Eine ausführliche Beschreibung der Funktionsweise von SableCC findet sich in [SableCC].

Neben dem Lexer, Parser und den Syntaxbaum-Knoten generiert SableCC das Package `quest.odl.parser.analysis` – hier befindet sich das Interface `Analysis`, das zusammen mit den Syntaxbaum-Knoten aus `quest.odl.parser.node` das *Visitor*-Entwurfsmuster implementiert ([GammaEtAl], S.301-318). Für jeden Syntaxbaum-Knoten `Foo`, d.h., für jede Unterklasse von `quest.odl.parser.node.Node`, enthält das Interface `Analysis` die Methode `caseFoo(Foo node)`. Jeder Syntaxbaum-Knoten `Foo` besitzt seinerseits die Methode `apply(Analysis visitor)`, die den Aufruf `visitor.caseFoo(this)` enthält.

Das Interface `Analysis` wird von der Klasse `AnalysisAdapter` implementiert, die für jeden Syntaxbaum-Knoten `Foo` eine leere Implementierung der entsprechenden Methode `caseFoo` bereitstellt. Die Unterklasse `DepthFirstAdapter` von `AnalysisAdapter` bildet die Vorstufe zur systematischen Analyse und Verarbeitung eines Syntaxbaums, indem sie einen Syntaxbaum in der Reihenfolge der Tiefensuche durchläuft. Dabei wird für jeden Knoten `Foo` beim Betreten die Methode `inFoo(Foo node)` und beim Verlassen die Methode `outFoo(Foo node)` aufgerufen. Diese Methoden führen hier keine Aktionen aus und dienen als Ansatzstellen für Unterklassen von `DepthFirstAdapter`, um Aktionen für Syntaxbaum-Knoten ausführen zu können. Wir wollen diese Vorgehensweise am Beispiel der Verarbeitung einer Gleichheit im `DepthFirstAdapter` erläutern. Für den Knoten `AEqualExpression`, der einen Gleichheitsausdruck repräsentiert, enthält `DepthFirstAdapter` die Methode

```
public void caseAEqualExpression(AEqualExpression node) {
    inAEqualExpression(node);

    node.getLeftExpression.apply( this );
    node.getRightExpression.apply( this );

    outAEqualExpression(node);
}
```

Auf diese Art und Weise wird für alle Knoten eines Syntaxbaums sichergestellt, dass sie in der Reihenfolge der Tiefensuche besucht werden, wobei für jeden Knoten beim Betreten und beim Verlassen eine Aktion ausgeführt werden kann. Um einen Syntaxbaum nun sinnvoll verarbeiten zu können, muss eine Unterklasse von `DepthFirstAdapter` erstellt werden, welche die Methoden `inFoo` und `outFoo` für Syntaxbaum-Knoten überschreibt, damit in diesen Methoden bestimmte Aktionen für die jeweiligen Knoten ausgeführt werden. Im einfachsten Fall wird in einer überschriebenen Methode `outFoo` ein ODL-Auswertungsbaum-Knoten erstellt, der dem verarbeiteten Syntaxbaum-Knoten entspricht. Für den Fall eines Gleichheitsterms würde die überschriebene Methode `outAEqualExpression(AEqualExpression node)`, vereinfacht dargestellt, wie folgt aussehen:

```
public void outAEqualExpression(AEqualExpression node) {
    /* Zunächst werden die von den Kinderknoten in einer
       Hashtabelle gespeicherten Auswertungsbaum-Knoten für
       die linke und rechte Seite der Gleichheit ausgelesen. */
    Term leftTerm = (Term)getOut(node.getLeftExpression());
    Term rightTerm = (Term)getOut(node.getRightExpression());

    /* Nun wird ein Auswertungsbaum-Knoten erstellt, der die
       Funktionalität einer Gleichheit bereitstellt. */
    Term equalExpression = new EqualExpression(leftTerm,rightTerm);

    /* Schließlich wird der neue Auswertungsbaum-Knoten in einer
       Hashtabelle abgespeichert, aus der er von anderen Knoten
       wieder ausgelesen werden kann. */
    setOut(node,equalExpression);
}
```

Die Klasse `quest.odl.evaluation.generator.SableCCGenerator`, die auf der Abbildung 5.2 der Komponente "`ODL Evaluation Tree Generator`" entspricht, ist im ODL-Auswertungssystem für die Erstellung eines ODL-Auswertungsbaums aus einem Syntaxbaum zuständig – sie erbt von der Klasse `DepthFirstAdapter` und überschreibt ihre Methoden derart, dass ein ODL-Syntaxbaum auf einen ODL-Auswertungsbaum abgebildet wird. Hierbei wird auch die semantische Korrektheit der verarbeiteten ODL-Abfrage überprüft, wie beispielsweise die Typkompatibilität des linken und rechten Operanden einer Gleichheit u.Ä.

Die Auswertung eines ODL-Auswertungsbaums wird veranlasst, indem die `evaluate`-Methode der Wurzel des Auswertungsbaums aufgerufen wird, die rekursiv die `evaluate`-Methoden für alle Kinderknoten aufruft und aus den zurückgelieferten Teilergebnissen das Auswertungsergebnis berechnet. Eine kurze schematische Darstellung der Struktur von Auswertungsbaum-Knoten wird auf der Abbildung 5.3 und in [Pasch] auf S.23 gegeben. Ausführliche Klassendiagramme befinden sich im Anhang A auf Abbildungen A.3 und A.4.

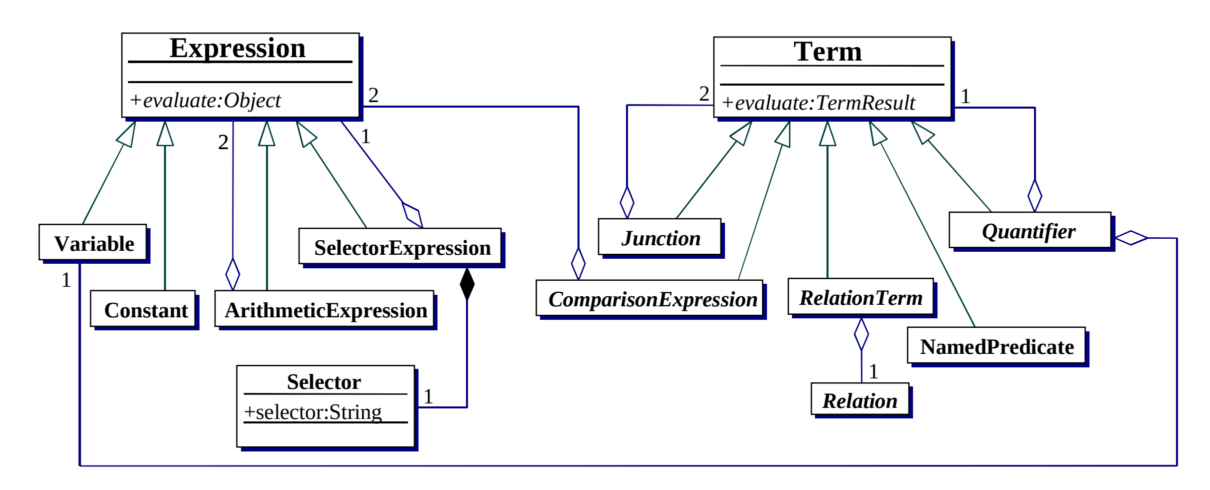


Abbildung 5.3: Grundbausteine eines ODL-Auswertungsbaums: Expression und Term

Die Grundbausteine des Auswertungsbaums sind alle Klassen, die eines der Interfaces `Term` und `Expression` implementieren. Alle Auswertungsbaum-Knoten – außer den Datentypen – erben von einem dieser Interfaces. Der grundsätzliche Unterschied zwischen einem Term und einer Expression besteht in Folgendem:

– Eine Expression dient zur Berechnung eines Ausdrucks und liefert als Ergebnis einen einzelnen Wert beliebigen Typs.
– Ein Term gibt als Ergebnis eine Instanz von `TermResult` zurück, die stets einen booleschen Wert als Ergebnis der Termauswertung und eventuelle weitere Daten enthält, wie die erfüllenden Variablenbelegungen, für die der Term das gewünschte boolesche Ergebnis liefert, bei der Auswertung des Terms erstellte neue Entitäten sowie Modifikationen an vorhandenen Entitäten.

Ein Term stellt also im Unterschied zu einer Expression einen logischen Ausdruck dar, der auch Quantoren und Relationsoperatoren enthalten darf [1].

Sowohl in der `Term`-Hierarchie, als auch in der `Expression`-Hierarchie findet das *Kompositum*-Entwurfsmuster ([GammaEtAl], S.239-253) Anwendung. Bei den Expressions spielen beispielsweise `SelectorExpression` und `ArithmeticExpression` die Rolle des Kompositums, bei den Termen sind es `Quantifier`, `Junction` und einige weitere Klassen.

Ausgehend von dem beschriebenen Aufbau des ODL-Abfrage-Verarbeitungssystems bedarf die

[1] Hier muss der Vollständigkeit halber auf die nicht immer konsequente Benennung von ODL-Auswertungsklassen hingewiesen werden. Einige Klassen, die in Wirklichkeit ein Term sind, enthalten die Bezeichnung `Expression` in ihrem Namen, wie beispielsweise die Klasse `EqualExpression`, die einen Term implementiert, der zwei Werte auf Gleichheit überprüft.

Einführung eines neuen Sprachkonstrukts oder die Änderung eines bestehenden Sprachkonstrukts dreier Schritte zur Implementierung:

1) In der SableCC-Grammatik von ODL muss die Produktionsregel für das neue Konstrukt eingeführt werden bzw. die bestehende Regel geändert werden.
2) Für ein neues Sprachkonstrukt muss eine Unterklasse von `Term` (falls es sich um einen ODL-Term handelt) oder `Expression` (falls es sich im einen ODL-Ausdruck handelt) im Package `quest.odl.evaluation.model` erstellt werden, die die geforderte Funktionalität implementiert. Handelt es sich bei dem neuen Sprachkonstrukt um einen Typ, so muss die entsprechende Auswertungsklasse das Interface `MetaType` (auf Abbildung 5.3 nicht gezeigt) implementieren. Für den Fall der Änderung eines bestehenden Sprachkonstrukts muss die entsprechende Klasse ggf. angepasst werden.
3) In der Generator-Klasse `SableCCGenerator` (Package `quest.odl.evaluation.generator`) müssen Methoden implementiert bzw. angepasst werden, die aus den Syntaxbaum-Knoten für das betreffende Sprachkonstrukt einen ODL-Term oder einen ODL-Ausdruck erstellen.

Die folgenden Abschnitte beschreiben nun die Grammatik- und die Programmänderungen, die zur Realisierung der geforderten Erweiterung des Sprachumfangs notwendig waren.

5.1.1 Änderung der ODL-Grammatik

In diesem Abschnitt beschreiben wir die Änderungen und Erweiterungen, die an der SableCC-Grammatik von ODL vorgenommen wurden. Wir werden für alle in 4.1 vorgestellten Sprachkonstrukte die entsprechenden Produktionsregeln angeben sowie die Änderungen im Vergleich zur vorherigen Version der ODL-Grammatik erklären.

Im Interesse der Lesbarkeit werden die Produktionen nicht immer streng nach den Regeln einer SableCC-Grammatik notiert – dies betrifft vor allem Sonderzeichen, die in einer SableCC-Grammatik in einem speziellen Abschnitt deklariert und später durch die ihnen zugewiesenen Schlüsselwörter ersetzt werden müssen. Beispielsweise werden wir die Produktion:

```
restricted_type_definition =
  l_brace variable colon type v_line ccl_proposition r_brace;
```

in der leichter zu lesenden Form

```
restricted_type_definition =
   { variable : type | ccl_proposition };
```

notieren.

Im Anhang C findet sich die vollständige ODL-Grammatik in der SableCC-Notation.

Nun wollen wir die Grammatik-Erweiterungen im Einzelnen erörtern.

- **Typsystem**
  Die erste Erweiterung der ODL-Grammatik stellt die Einführung neuer Datentypen dar – dies sind `ProductType`, `RestrictedType`, `SetType` sowie der Sondertyp `IntroducedType`, der den Zugriff auf Elemente einer Menge realisiert.

  Die alten Produktionen für Datentypen erzeugten die vier Typen `Boolean`, `Integer`, `String` und `Entity` (in der Grammatik `model_element_type`):

```
bool_type   = 'boolean';
int_type    = 'int';
string_type = 'String';

type = basic_type |
       model_element_type;
```

```
basic_type = bool_type |
             int_type |
             string_type;

model_element_type = identifier;
```

Die neuen Produktionen spiegeln das komplizierter gewordene Typensystem wider. Als Erstes wird die Gruppe der `unären` Typen herausgesondert, die Grundtypen und Entitäten enthält, also die Typen, deren Instanzen aus genau einem Wert bestehen:

```
unary_type = basic_type |
             model_element_type;

basic_type = bool_type |
             int_type |
             string_type;

model_element_type = identifier;
```

Der Produkttyp wird nun mit den folgenden Produktionen eingeführt:

```
product_type = unary_type |
               ( type_list );

type_list = identifier : type type_list_tail*;

type_list_tail = , identifier : type;
```

Dabei ist zu beachten, dass unäre Typen in der Grammatik als Spezialfall eines Produkttyps deklariert werden, während in der späteren Implementierung zwischen unären Typen und Produkttypen deutlich unterschieden wird, indem sie durch verschiedene Klassen implementiert werden.

Als nächster Schritt wird der eingeschränkte Typ eingeführt:

```
restricted_type_definition = { variable : type | ccl_proposition };
```

Auf die hier verwendete Produktion `ccl_proposition` wird weiter unten eingegangen.

Der Mengentyp wird nun durch die Produktion

```
set_type_definition = set type;
```

eingeführt.

Schließlich wird die Ausgangsproduktion für alle Typen definiert:

```
type = product_type |
       restricted_type_definition |
       set_type_definition |
       element defined_variable |
       element ( expression );

defined_variable = variable;

variable = identifier;
```

Die Deklaration des `IntroducedType` ist direkt in der Produktionsregel für `type` untergebracht, und zwar in den Produktionen

```
type = element defined_variable |
       element ( expression );
```

Ausgehend von dieser Definition werden für `IntroducedType` zunächst alle Ausdrücke nach dem Schlüsselwort `element` akzeptiert. Erst bei der semantischen Analyse durch den `SableCCGenerator` wird überprüft, ob der Ausdruck nach dem Schlüsselwort `element` ein mengenwertiges Ergebnis liefert.

- **Einschränkung beim `new`-Quantor**
  Ein `new`-Quantor dient dazu, eine neue Entität zu erzeugen – die von ihm gebundene Variable muss also stets vom Typ `model_element_type` sein. Diese Einschränkung wurde im Zuge der Entwicklung der Grammatik von der semantischen Ebene des `SableCCGenerator`'s auf die syntaktische Ebene der Grammatik verlagert, indem die frühere Produktionsregel

```
unary_proposition =
  neg unary_proposition |
  quantifier variable_definition . unary_proposition |
  term;

variable_definition = variable : type;
```

  durch die folgende Regel ersetzt wurde:

```
unary_proposition =
  neg unary_proposition |
  quantifier variable_definition . unary_proposition |
  new_quantifier model_element_variable_definition .
    unary_proposition |
  term;

variable_definition = variable : type;

model_element_variable_definition = variable : model_element_type;
```

  Die Angabe eines ungeeigneten Variablentyps für einen `new`-Quantor wird nach dieser Änderung bereits während der Syntaxanalyse einer ODL-Abfrage entdeckt.

- **Benamte Prädikate**
  Um die benamten Prädikate in die Grammatik einzuführen, wurde ein neues Startsymbol hinzugefügt, der entweder die Ableitung für eine übliche ODL-Abfrage oder eine Ableitung für ein benamtes Prädikat erzeugt:

```
odl_start = proposition |
            named_predicate_declaration;
```

  Hierbei ist `proposition` das frühere Startsymbol, aus dem jede ODL-Abfrage abgeleitet werden kann, die kein benamtes Prädikat definiert.

  Ein benamtes Prädikat wird durch die folgende Produktionsregel definiert:

```
named_predicate_declaration =
  identifier(type_list) := ccl_proposition;
```

Damit ein früher definiertes benamtes Prädikat aus einer ODL-Abfrage heraus aufgerufen werden kann, wurde in die Produktion `unary_proposition` folgende Regel eingefügt:

```
unary_proposition = ... |
                    named_predicate_call |
                    ...;

named_predicate_call = call call_expression;
```

wobei `call` ein neues Schlüsselwort ist und die Produktion `call_expression`, die bereits früher in der Grammatik vorhanden war, einen Funktionsaufruf der Form `ident(argumentList)` definiert.

- **CCL-Propositionen**
  Wie bereits früher angesprochen, wird in der neuen Grammatik an einigen Stellen, wie beispielsweise bei der Definition eingeschränkter Typen, nur die Verwendung von CCL-Propositionen zugelassen, die im Vergleich zu ODL-Propositionen folgende Einschränkungen aufweisen:
  - Es dürfen nur die Quantoren `exists` und `forall` verwendet werden. Die Quantoren `context` und `new` sind nicht zulässig.
  - Das Schlüsselwort `result` darf nicht verwendet werden. Dadurch wird ausgeschlossen, dass Attribute oder Assoziationen von Entitäten modifiziert werden können.

  Diese Einschränkungen müssen in der Grammatik dadurch implementiert werden, dass die Produktionen für CCL-Propositionen weitgehend parallel zu den Produktionen für ODL-Propositionen aufgebaut werden, wobei sich die Einschränkungen darin niederschlagen, dass bestimmte Ableitungen im CCL-Zweig nicht vorhanden sind.

  Wir notieren die Produktionsregeln für ODL-Propositionen und CCL-Propositionen in zwei Spalten nebeneinander, damit die Ähnlichkeiten und die Unterschiede zwischen den beiden Zweigen leichter zu sehen sind.

Produktionen für ODL-Propositionen

```
proposition =
 unary_proposition |
 proposition and
   unary_proposition |
 proposition or
   unary_proposition |
 proposition implies
   unary_proposition |
 proposition equiv
   unary_proposition;


unary_proposition =
 neg unary_proposition |
 named_predicate_call |
 term |
 quantifier
   variable_definition .
   unary_proposition |
 new_quantifier
   model_element_variable_definition
   . unary_proposition;
```

Produktionen für CCL-Propositionen

```
ccl_proposition =
 ccl_unary_proposition |
 ccl_proposition and
   ccl_unary_proposition |
 ccl_proposition or
   ccl_unary_proposition |
 ccl_proposition implies
   ccl_unary_proposition |
 ccl_proposition equiv
   ccl_unary_proposition;


ccl_unary_proposition =
 neg ccl_unary_proposition |
 named_predicate_call |
 ccl_term |
 ccl_quantifier
   variable_definition .
   ccl_unary_proposition;
```

```
ccl_quantifier =
 forall |
 exists;

quantifier =
 ccl_quantifier |
 context;

new_quantifier =
 new;


term =
  basic_proposition |
  ( proposition );


basic_proposition =
 relation |
 bool_proposition;


relation =
 pre_relation |
 post_relation;


pre_relation =
 is call_expression;

post_relation =
 result has call_expression |
 result not has call_expression;
```

```
ccl_quantifier =
 forall |
 exists;


ccl_term =
 ccl_basic_proposition |
 ( ccl_proposition );


ccl_basic_proposition =
 ccl_relation |
 bool_proposition;


ccl_relation =
 pre_relation;


pre_relation =
 is call_expression;
```

- **Selektoren**
  Die Produktionsregeln für Selektoren wurden bereits in der früheren Grammatik eingeführt, sodass sie für die nun vorgenommene Implementierung von Selektorausdrücken nicht mehr eingefügt, sondern nur geringfügig angepasst werden mussten. Da Selektorausdrücke jedoch nicht im Funktionsumfang der früheren Version des ODL-Interpreters enthalten waren, erzeugte der `SableCCGenerator` eine `UnsupportedConstructException`, wenn er auf einen Selektorausdruck im verarbeiteten ODL-Syntaxbaum traf. Die eigentliche Implementierung der Selektorausdrücke fand also im `SableCCGenerator` statt, und wird im Abschnitt 5.1.2 besprochen.

- **Vergleiche**
  In Analogie zum bereits vorhandenen Test auf Gleichheit für zwei Ausdrücke

  ```
  equal_expression =
    expression = expression;
  ```

  wurden die Produktionen für Größer-Kleiner-Vergleiche zweier Ausdrücke eingeführt

  ```
  bigger_smaller_expression =
    expression comparison_operator expression;
  ```

```
comparison_operator =
  > |
  < |
  >= |
  <=;
```

Die Unterscheidung zwischen Gleichheitstest und Vergleich auf der syntaktischen Ebene wird vorgenommen, weil eine Unterscheidung zwischen Gleichheitstest und Vergleich auf der semantischen Ebene vorhanden ist: auf Gleichheit können zwei Werte beliebigen Typs getestet werden, während der Vergleich nur für Werte möglich ist, die das Interface `java.lang.Comparable` implementieren – im ODL-Typsystem sind es nur `Int` und `String`.

- **Arithmetische Ausdrücke**
  Als Grundlage für die Produktionsregeln für arithmetische Ausdrücke wurden die Produktionen aus [SableCC] (S.26) genommen, die in ähnlicher Form auch in Compilerbau-Lehrbüchern zu finden sind. Da ODL zurzeit keine Division unterstützt, wurde die entsprechende Ableitung ausgenommen. Die Produktionsregeln für arithmetische Ausdrücke lauten:

```
arithmetic_expression =
  factor |
  arithmetic_expression + factor |
  arithmetic_expression - factor;

factor =
  arithmetic_term |
  factor * arithmetic_term;

arithmetic_term =
  int_constant_expr |
  ( arithmetic_expression ) |
  size ( expression ) ;

int_constant_expr = sign? int_constant;

sign =
  + |
  -;
```

  In der Produktion `arithmetic_term` wurden alle ODL-Funktionen zusammengefasst, die ein numerisches Ergebnis zurückliefern: zurzeit ist es nur `size( expression )`, das die Größe einer Menge berechnet. Sollten zukünftig weitere Funktionen mit numerischem Ergebnis in den Sprachumfang aufgenommen werden, dann müssten sie als Ableitungen von `arithmetic_term` in die ODL-Grammatik integriert werden.

  Das Grammatik-Symbol, aus dem ein arithmetischer Ausdruck abgeleitet werden kann, ist `expression`. Da die Produktionsregeln für Ausdrücke bei der Einführung arithmetischer Ausdrücke verändert werden mussten, um die Grammatik – wie von SableCC verlangt – LL(1)-konform zu gestalten, beschreiben wir an dieser Stelle die Produktionsregeln für Ausdrücke etwas ausführlicher:

```
expression =
  non_constant_expression |
  arithmetic_expression |
  constant_expression;
```

```
non_constant_expression =
  functional_expression |
  selector_expression |
  defined_variable;

constant_expression =
  bool_constant_expr |
  string_constant_expr;
```

Es wurde also eine Unterscheidung zwischen konstanten Ausdrücken, nicht-konstanten Ausdrücken und arithmetischen Ausdrücken gemacht, wobei ganzzahlige Konstanten zu den arithmetischen Ausdrücken und nicht zu den konstanten Ausdrücken gehören (andernfalls ließ sich keine Grammatik erstellen, die von SableCC verarbeitet werden kann). Zu den konstanten Ausdrücken gehören boolesche Konstanten und Zeichenketten, zu den nicht-konstanten Ausdrücken zählen Selektorausdrücke, deklarierte Variablen und funktionelle Ausdrücke. Letztere dürfen zurzeit noch nicht eingesetzt werden – bei ihrer Verwendung wird vom `SableCCGenerator` eine `UnsupportedConstructException` erzeugt.

- **Erweiterte Syntax des `context`-Quantors**
  Für den `context`-Quantor wurde eine Erweiterung eingeführt, über die optionale Parameter spezifiziert werden können, die das Aussehen des Eingabedialogs für die vom `context`-Quantor gebundene Variable beeinflussen. Zurzeit ist nur die Angabe eines Hinweistextes für den Eingabedialog möglich.

```
quantifier =
  ccl_quantifier |
  context context_extension?;

context_extension =
  [ hint_extension? ];

hint_extension =
  hint = string_constant_expr_list;
```

Hierbei ist `string_constant_expr_list` eine durch Kommata getrennte Liste aus einer oder mehreren Zeichenketten. Sie wird wie folgt definiert:

```
string_constant_expr_list =
  string_constant_expr string_constant_expr_list_tail*;

string_constant_expr_list_tail =
  , string_constant_expr;
```

5.1.2 Implementierung erweiterter und neuer Sprachkonstrukte

Dieser Abschnitt stellt die Implementierung der Erweiterung des ODL-Sprachumfangs im ODL-Auswertungssystem dar. Wir werden die Änderungen an vorhandenen ODL-Auswertungsklassen sowie neue ODL-Auswertungsklassen beschreiben. Des Weiteren zeigen wir die Änderungen am `SableCCGenerator`, die notwendig waren, um die neuen ODL-Auswertungsklassen einsetzen zu können.

- **Typsystem**
  Das ODL-Typsystem ist vom Java-Typsystem soweit wie möglich abgekoppelt und hat eigene Typ-Klassen, die den Typ eines Werts beschreiben. Ein Typklassen-Name wird nach

dem Schema Meta_Typbeschreibung zusammengesetzt: String-Werte haben beispielsweise den ODL-Typ MetaString und Int-Werte den ODL-Typ MetaInteger. Metamodell-Elemente werden als *Entitäten* verstanden und haben dementsprechend den ODL-Typ MetaEntity. Hat ein ODL-Typ keine Entsprechung im Java-Typsystem, so wird eine spezielle *Werteklasse* erstellt, um Instanzen dieses Typs zu repräsentieren – für den ODL-Typ MetaEntity werden die Werte durch Instanzen der Klasse Entity repräsentiert.

Um der umfassenden Erweiterung des ODL-Typsystems gerecht zu werden, die im Rahmen der vorliegenden Diplomarbeit vorzunehmen war, wurde die MetaType-Klassensierarchie umgebaut.

Zunächst wurde für alle MetaType-Unterklassen die abstrakte Oberklasse AbstractMetaType erstellt, die einige Methoden aus dem Interface MetaType implementiert – alle MetaType-Unterklassen sind auch Unterklassen von AbstractMetaType. Die Klasse MetaProductType nimmt eine Sonderstellung in der MetaType-Hierarchie ein, indem sie nicht direkt das Interface MetaType, sondern das Interface MetaCompositeType implementiert, das ein Unterinteface von MetaType ist und Methoden für zusammengesetzte Datentypen definiert, d.h., Datentypen, die aus Wertetupeln fester Länge bestehen. Die abstrakte Klasse AbstractMetaCompositeType implementiert einige Methoden aus MetaCompositeType und dient als Oberklasse für alle Klassen, die einen zusammengesetzten Typ implementieren – zurzeit ist MetaProductType ihre einzige Unterklasse.

Die vollständige MetaType-Hierarchie ist auf der Abbildung A.5 im Anhang A dargestellt.

Wir beschreiben nun die Erweiterungen im Detail. In der früheren Version waren das Interface MetaType sowie die Typ-Klassen MetaBool, MetaInteger, MetaString und MetaEntity bereits vorhanden, ebenso wie die Werte-Klasse Entity, die einzelne Metamodell-Elemente repräsentiert. Für die neuen Datentypen ProductType, RestrictedType, SetType und IntroducedType mussten neue Typ-Klassen und teilweise auch Werte-Klassen erstellt werden.

Die folgende Tabelle gibt für jeden neuen Datentyp die dafür erstellten Klassen in der MetaType-Hierarchie sowie die implementierten Methoden im SableCCGenerator an. Ein ausführlicher Kommentar zu diesen Klassen und Methoden findet sich im Java-Quellcode der entsprechenden Klassen und Methoden sowie in [ODLAPI].

Datentyp	**Typ-Klassen**	**Methoden im SableCCGenerator**
ProductType (Produkttyp)	MetaCompositeType, AbstractMetaCompositeType, MetaProductType, CompositeValue, ProductValue	outAProductType, outAListProductType, constructMetaProductType, outATypeList
SetType (Mengentyp)	MetaSetType, SetValue	outASetType, outASetTypeDefinition
RestrictedType (Eingeschränkter Typ)	MetaRestrictedType	outARestrictedType, inARestrictedTypeDefinition, caseARestrictedTypeDefinition, outARestrictedTypeDefinition
IntroducedType (Eingeführter Typ)	MetaIntroducedType	outASetExpressionType, outASetVariableType, constructMetaIntroducedType

Für alle zusammengesetzten Typen, die das Interface MetaCompositeType implementie-

ren, wurde die abstrakte Werte-Klasse `CompositeValue` eingeführt. Ihre Unterklasse `ProductValue` stellt Werte des Typs `MetaProductType` dar: sie enthält ein Tupel von Werten, deren Typen genau den spezifizierten Elementtypen im zugeordneten `MetaProductType` entsprechen müssen. Hierbei spielt die Klasse `MetaCompositeType` die Rolle des *Kompositums* ([GammaEtAl], S.239-253)) in der `MetaType`-Hierarchie und `CompositeValue` spielt die Rolle des Kompositums für Werteklassen (da für Werteklassen keine eigene Oberklasse festgelegt wurde, tritt `java.lang.Object` als Oberklasse für alle Werteklassen auf).

Für den Typ `MetaSetType` die Werteklasse `SetValue` erstellt: eine `SetValue`-Instanz enthält eine Kollektion von beliebig vielen Werten des Basistyps der Menge, der im zugeordneten `MetaSetType` spezifiziert wird.

Eine wesentlichen Funktionalität der `MetaType`-Unterklassen ist die Rückgabe eines Iterators über die Werte des repräsentierten Typs (der zurückgegebene Iterator muss das Interface `java.util.Iterator` implementieren). Dafür wird im Interface `MetaType` die Methode `Iterator instances` definiert . In `MetaBool` liefert sie beispielsweise einen Iterator über die Kollektion `{false, true}`.

Für die Implementierung der `instances`-Methode bei den Typen `MetaProductType`, `MetaRestrictedType` und `MetaSetType` wurden Hilfsklassen im Package `quest.util.collections` erstellt, die wir hier kurz beschreiben:

– `ResettableIterator`
  Interface, das zusätzlich zu den Methoden des Interfaces `java.util.Iterator` die Methode `reset()` definiert, die den Iterator auf den Beginn der iterierten Kollektion zurücksetzt.

– `CachedIterator`
  Implementierung von `ResettableIterator`. Ein `CachedIterator` kapselt einen Iterator über die Zielkollektion und leitet die Aufrufe der Methoden `next` und `hasNext` anfangs an diesen Iterator weiter. Gleichzeitig speichert er alle zurückgegebenen Werte in einer internen Liste. Wird der `CachedIterator` mit einem `reset`-Aufruf auf den Anfang der Kollektion zurückgesetzt, so greift er auf die Cache-Liste zu, um bereits ausgelesene Werte wieder zurückliefern zu können.

  Ein wesentlicher Vorteil dieser Klasse ist, dass sie nur beim ersten Durchlauf durch die Kollektion auf den Kollektionsiterator zugreifen muss, der unter Umständen wesentlich langsamer als ein einfacher Listeniterator sein kann. Bei den nachfolgenden Durchläufen durch die Kollektion kann der Zugriff auf ihre Elemente wesentlich schneller erfolgen, weil sie nun lediglich aus der Cache-Liste geholt werden müssen.

– `CompositeIterator`
  Iteriert über das kartesische Produkt mehrerer Kollektionen. Implementiert ebenfalls das Interface `ResettableIterator`. Um die Funktionsweise dieses Iterators zu verdeutlichen, nehmen wir als Beispiel die Kollektionen `(A,B,C)` und `(1,2)`:
  für sie würde ein `CompositeIterator` folgende Werte in Form von `Object`-Arrays der Länge 2 liefern: `[A,1], [A,2], [B,1], [B,2], [C,1], [C,2]`.

– `ConditionedIterator`
  Kapselt einen Kollektionsiterator und wendet als Filter für die Kollektionselemente eine Bedingung an, die über das Interface `ConditionedIterator.Condition` spezifiziert werden kann, indem die Methode `boolean meetsCondition( Object value )` implementiert wird. Ein `ConditionedIterator` gibt genau die Elemente der gekapselten Kollektion zurück, die die spezifizierte Bedingung erfüllen. Für die anzuwendende Bedingung gibt es dabei keine Einschränkungen, außer dass die Methode `meetsCondition( Object value )` immer terminieren und ein boolesches Ergebnis zurückgeben soll.

– `PowersetIterator`
  Iteriert über die Potenzmenge einer spezifizierten Kollektion. Für die Kollektion
  `(A,B,C)`
  würde ein `PowersetIterator` beispielsweise folgende Mengen in Form von `java.util.LinkedList`-Instanzen liefern:
  `({}, {A}, {B}, {A,B}, {C}, {A,C}, {B,C}, {A,B,C})`.
  Die Parameterkollektion wird, wie auch bei anderen Iteratoren aus diesem Package, über ihren Iterator spezifiziert.

Ein ausführlicher Kommentar zu den Klassen im Package `quest.util.collections` befindet sich im Quellcode und in [ODLAPI]. Ein Klassendiagramm des Packages wird im Anhang A auf der Abbildung A.7 gezeigt.

Da die oben beschriebenen Klassen im Zuge der Implementierung von `MetaType`-Unterklassen für die neuen ODL-Datentypen erstellt wurden, wollen wir angeben, welche Datentypen von welchen Iteratoren Gebrauch machen, um in ihrer `instances`-Methode einen Iterator über die Werte des jeweiligen Typs zu konstruieren:

ODL-Datentyp	**Klasse**	**Benutzte Klassen aus `quest.util.collections`**
Produkttyp	`MetaProductType`	`CompositeIterator`, `ResettableIterator`
Eingeschränkter Typ	`MetaRestrictedType`	`ConditionedIterator` `ConditionedIterator.Condition` `ResettableIterator`
Mengentyp	`MetaSetType`	`PowersetIterator` `ResettableIterator`

An dieser Stelle ist ein Hinweis zur Gestaltung effizienter ODL-Abfragen notwendig: die (indirekte) Benutzung der Klasse `CachedIterator` bei der Iteration durch Tupel eines Produkttyps (indem `MetaProductType` auf die Klasse `CompositeIterator` zurückgreift, die ihrerseits `CachedIterator` benutzt) hat den Vorteil zur Folge, dass eine Abfrage der Form

```
exists var:( p1:Port, p2:Port, c:Component ).(
  var.p1.Component = var.c and var.p2.Component = var.c )
```

durch das Caching zum Teil erheblich schneller ausgewertet wird, als die Abfrage

```
exists p1:Port. exists p2:Port. exists c:Component.(
  p1.Component = c and p2.Component = c ).
```

Dies sollte beim Entwurf und Optimierung von ODL-Abfragen berücksichtigt werden. Mehr Informationen zum Entwurf effizienter Abfragen gibt es im Abschnitt 5.4.

- **Benamte Prädikate**
  Für die benamten Prädikate waren zwei Anwendungsfälle zu implementieren:

  1) Deklaration eines benamten Prädikats. Beispiel:
  ```
  compHasSubComps( comp:Component ) :=
    size( comp.SubComponents ) > 0
  ```
  2) Aufruf eines zuvor deklarierten benamten Prädikats. Beispiel:
  ```
  exists c:Component. call compHasSubComps( c )
  ```

Wir beschreiben nun die Implementierung beider Anwendungsfälle.

– Deklaration eines benamten Prädikats
  Betrachten wir zunächst die Deklaration eines benamten Prädikats. Die semantische Analyse und die Kompilation des ODL-Terms, der den Rumpf eines benamten Prädikats bildet, unterscheidet sich nicht von der Verarbeitung eines ODL-Terms in einer üblichen ODL-Abfrage. Der Unterschied tritt erst bei der Behandlung des kompilierten ODL-Terms zu Tage – er wird vom `SableCCGenerator` nicht als Ergebnis der Kompilation an das aufrufende Objekt (typischerweise eine Instanz von `quest.odl.evaluation.EvaluationModelGenerator`) zurückgegeben, sondern in einer Instanz der neuen Klasse `NamedPredicate` gekapselt und in einer internen Hashtabelle unter dem in der Prädikatsdeklaration angegebenen Namen für spätere Aufrufe gespeichert. Für diese Sonderbehandlung sind im `SableCCGenerator` die Methoden `outANamedPredicateOdlStart`, `outANamedPredicateDeclaration` und `caseANamedPredicateDeclaration` zuständig. Eine `NamedPredicate`-Instanz enthält alle für den Aufruf eines benamten Prädikats notwendigen Angaben – die Parameterliste des benamten Prädikats und seinen Rumpf, der durch einen ODL-Term repräsentiert wird.

  Diese Verarbeitungsweise hat den positiven Nebeneffekt, dass benamte Prädikate, die im ODL-Editor während einer QUEST-Sitzung deklariert wurden, bis zum Ende dieser Sitzung verfügbar bleiben, ohne erneut eingegeben werden zu müssen.

– Aufruf eines benamten Prädikats
  Wir beschreiben nun die Implementierung des Aufrufs eines benamten Prädikats. Dafür wird die Klasse `NamedPredicateTerm` eingesetzt: sie erfüllt die Rolle eines Adapters (s. auch [GammaEtAl], S.171-185), der einer `NamedPredicate`-Instanz die Schnittstelle eines Terms gibt. Diese Trennung zwischen `NamedPredicate` und `NamedPredicateTerm` wird gemacht, um die Auswertung eines benamten Prädikats von der Berechnung seiner Parameter abzukoppeln, die ja beliebige ODL-Ausdrücke sein dürfen. Die Auswertung der Parameter wird von `NamedPredicateTerm` vorgenommen und die Ergebnisse werden als Parameterwerte an den `NamedPredicate` weitergegeben.

  Im `SableCCGenerator` sind die Methoden `outANamedPredicateUnaryProposition`, `outANamedPredicateCclUnaryProposition` und `outANamedPredicateCall` für die Erstellung des Aufrufs eines benamten Prädikats zuständig. Die ersten beiden dienen lediglich dazu, einen bereits erstellten `NamedPredicateTerm` weiterzugeben. Die Erstellung und Überprüfung eines `NamedPredicateTerm` findet in der Methode `outANamedPredicateCall` statt. Hier wird zunächst überprüft, ob das aufgerufene Prädikat bereits deklariert wurde, dann wird überprüft, ob die Anzahl und die Typen der Parameter mit der deklarierten Signatur des benamten Prädikats übereinstimmen. Schließlich wird ein `NamedPredicateTerm` erstellt, der das aufzurufende `NamedPredicate` kapselt.

- **CCL-Propositionen**
  Für die CCL-Propositionen existiert in der ODL-Grammatik ein eigener Ableitungszweig, der mit einigen Ausnahmen mit dem für ODL-Propositionen übereinstimmt (s. Abschnitt 5.1). Da die Verarbeitung von Syntaxbaum-Knoten für eine CCL-Proposition und eine ODL-Proposition gleich ist, wurden im `SableCCGenerator` die Methoden für CCL-Propositionen erstellt, indem entsprechende Methoden für ODL-Propositionen kopiert und die Namen der Methoden und der in Methodenrümpfen aufgerufenen Methoden angepasst wurden. Beispielsweise wurde aus der Methode für die `AND`-Verknüpfung von zwei ODL-Propositionen:

```
public void outAAndProposition(AAndProposition node) {
  Term leftTerm = (Term)getOut( node.getProposition() );
  Term rightTerm = (Term)getOut( node.getUnaryProposition() );
  setOut( node, new Conjunction(leftTerm, rightTerm) );
}
```

durch Kopieren und Namensanpassung die Methode für die `AND`-Verknüpfung von zwei CCL-Propositionen erstellt:

```
public void outAAndCclProposition(AAndCclProposition node) {
  Term leftTerm = (Term)getOut( node.getCclProposition() );
  Term rightTerm = (Term)getOut( node.getCclUnaryProposition() );
  setOut( node, new Conjunction(leftTerm, rightTerm) );
}
```

Auf diese Art und Weise wurden folgende Methoden erstellt:

Methode	**Aufgabe**
`outAAndCclProposition`, `outAOrCclProposition`, `outAImpliesCclProposition`, `outAEquivCclProposition`, `outAUnopCclProposition`	CCL-Propositionen
`outANegCclUnaryProposition`, `outANamedPredicateCclUnaryProposition`, `outATermCclUnaryProposition`, `caseAQuantifierCclUnaryProposition`	Unäre CCL-Propositionen
`outARelationCclBasicProposition`, `outABoolCclBasicProposition`	CCL-Grundpropositionen
`outAIsCclRelation`	CCL-Relationen
`outABasicCclTerm`, `outAParCclTerm`	CCL-Terme

Eine weitere Besonderheit musste im Zusammenhang mit der Verwendung von CCL-Propositionen für eingeschränkte Typen beachtet werden. Da CCL-Propositionen, die als Restriktionsbedingung für eingeschränkte Typen verwendet werden, wie beispielsweise die Bedingung `size(comp.Ports)>2` in der Abfrage

`exists component:{ comp:Component | size(comp.Ports)>2 }.true`

einen eigenen Namensraum für Variablen haben, muss neben einer bereits vorhandenen Tabelle für globale Variablennamen eine weitere Tabelle mit Variablennamen für CCL-Propositionen verwaltet werden. Hierfür wurden im `SableCCGenerator` in Analogie zu den Methoden `recordVariable` und `recordVariables`, die Variablen im globalen Namensraum abspeichern, die Methoden `recordCclVariable` und `recordCclVariables` erstellt, die Variablen im CCL-Namensraum abspeichern. Dementsprechend müssen die Methoden `casePCclQuantifierDefault` und `outADefinedVariable`, die auf Namensräume zugreifen und sie verändern, bei jedem Zugriff feststellen, ob zurzeit der globale Namensraum oder der lokale Namensraum einer CCL-Proposition aktiv ist. Die Umschaltung zwischen dem globalen und dem CCL-Namensraum findet in den Methoden `inARestricted-TypeDefinition` und `outARestrictedTypeDefinition` statt, weil der lokale CCL-Namensraum bislang nur für die Verarbeitung von eingeschränkten Typen benutzt wird. Die erste Methode schaltet auf den CCL-Namensraum um, und die zweite aktiviert wieder den globalen Namensraum. Die Methode `caseARestrictedTypeDefinition` schreibt den lokalen Variablennamen des eingeschränkten Typs (in der obigen Beispielabfrage ist es `comp`) in den CCL-Namensraum, um ihn für die Restriktionsbedingung verfügbar zu machen.

- **Selektoren**
  Ein Selektor kann in einer ODL-Abfrage in vier Fällen eingesetzt werden:

- Zugriff auf ein Element eines zusammengesetzten Typs (zurzeit nur durch Produkttypen vertreten).
- Zugriff auf ein Attribut einer Entität. Beispiel: Name einer Entität.
- Zugriff auf eine Assoziation zu anderen Entitäten. Hier sind zwei Fälle zu unterscheiden:
  - Zugriff auf eine Assoziation zu höchstens einer anderen Entität. Der zurückgegebene Wert ist eine Entität. Beispiel: Ausgangsport eines Kanals.
  - Zugriff auf eine Assoziation zu mehreren anderen Entitäten. Der zurückgegebene Wert ist eine Menge von Entitäten. Beispiel: Ports einer Komponente.

In allen vier Fällen müssen, trotz gleicher ODL-Syntax, verschiedene Auswertungsmechanismen zum Einsatz kommen. Daher gibt es vier verschiedene *Selektorklassen*, jede von denen einen der Mechanismen implementiert. Die nachfolgende Tabelle gibt die vier Selektorklassen an:

Selektor-Klasse	**Anwendung**
`CompositeTypeSelector`	Zugriff auf ein Element eines zusammengesetzten Typs.
`AttributeSelector`	Zugriff auf ein Attribut einer Entität.
`AssociatedEntitySelector`	Zugriff auf eine assoziierte Entität.
`AssociatedEntitiesSelector`	Zugriff auf meherere assoziierte Entitäten.

Jede Selektorklasse erbt von der abstrakten Klasse `Selector`. Ein Selektor an sich ist kein selbständiger ODL-Ausdruck – er wird von `SelectorExpression`-Instanzen benutzt. Jeder ODL-Selektorausdruck wird durch eine Instanz der Klasse `SelectorExpression` dargestellt. Dabei braucht sie nicht zu wissen, welcher Zugriffsmechanismus im Falle des konkreten Selektorausdrucks anzuwenden ist, weil der eigentliche Zugriff durch die Instanz einer Unterklasse von `Selector` realisiert wird, die der `SelectorExpression`-Instanz mitgegeben wird. Das entsprechende Klassendiagramm findet sich auf der Abbildung A.3 im Anhang A.

Für die Kompilation von Selektorausdrücken sind im `SableCCGenerator` folgende Methoden zuständig: `outASelectorExpression`, `outASelection`, `outASelector`. Die Entscheidung, welche der vier Selektorklassen in einem konkreten Fall zum Einsatz kommt, wird in der Methoden `outASelectorExpression` getroffen.

- **Arithmetische Ausdrücke**
  Im Zuge der Implementierung der arithmetischen Ausdrücke musste für jede eingeführte arithmetische Operation eine Klasse erstellt werden, die diese Operation ausführt:

Klasse	**Arithmetische Operation**
`AdditionExpression`	Addition zweier ganzen Zahlen.
`SubtractionExpression`	Subtraktion zweier ganzen Zahlen.
`MultiplicationExpression`	Multiplikation zweier ganzen Zahlen.

Alle drei Operationen sind binär, sodass es nahelag, eine gemeinsame abstrakte Oberklasse `BinaryArithmeticExpression` zu erstellen. Sie erledigt alle Aufgaben einer allgemeinen binären Operation (darunter die Verwaltung der beiden Operanden) und ruft in der `evaluate`-Methode, die das Ergebnis der Operation berechnen soll, die abstrakte Methode `calculateArithmeticExpression` auf. Hier wird also das Entwurfsmuster *Schablonenmethode* ([GammaEtAl], S.366-372) implementiert. Um eine binäre arithmetische Ope-

ration zu implementieren, muss nun lediglich eine Unterklasse von `BinaryArithmeticExpression` erstellt werden, die in der Methode `calculateArithmeticExpression` die notwendige arithmetische Operation ausführt. Genau dies tun die Klassen `AdditionExpression`, `SubtractionExpression` und `MultiplicationExpression`. Das entsprechende Klassendiagramm ist auf der Abbildung A.3 zu sehen.

Im `SableCCGenerator` sind folgende Methoden für die Verarbeitung von arithmetischen Ausdrücken verantwortlich: `outAConstantArithmeticTerm`, `outAExpressionArithmeticTerm`, `outAArithmeticTermFactor`, `outAMultFactor`, `outAFactorArithmeticExpression`, `outAPlusArithmeticExpression` und `outAMinusArithmeticExpression`.

Neben den oben genannten arithmetischen Operationen wurde der Ausdruck `size( setExpression)` implementiert, der die Größe einer Menge berechnet. Dieser Ausdruck ist zwar kein echter arithmetischer Ausdruck, sondern eine Funktion, er wurde aber in der ODL-Grammatik über die Produktion `arithmetic_term` eingeführt, weil er, wie auch arithmetische Ausdrücke, ein ganzzahliges Ergebnis zurückliefert. Der Ausdruck wird durch die Klasse `SetSizeExpression` implementiert, die einen Ausdruck als Parameter erhält, und die Größe der Menge berechnet, die der Parameterausdrück als Ergebnis liefert. Da dieser Ausdruck nur auf Mengen angewandt werden darf, wird bei seiner Kompilation im `SableCCGenerator` in der Methode `outASetSizeArithmeticTerm` überprüft, ob der Parameterausdruck immer ein mengenwertiges Ergebnis zurückgibt.

- **Vergleiche**
  In der früheren Version von ODL war für die Auswertung von Gleichheitstermen bereits die Klasse `EqualExpression`[2] vorhanden, die für zwei Ausdrücke beliebigen (aber gleichen) Datentyps feststellte, ob sie ihre Ergebnisse gleich sind. Mit den arithmetischen Ausdrücken wurden auch die Vergleichsterme eingeführt, die für zwei Ausdrücke (ebenfalls gleichen Typs) ermitteln, ob der erste größer oder kleiner als der zweite ist.

  Im Unterschied zum Gleichheitsterm können bei Vergleichstermen nur Werte der Typen `Int` und `String` verglichen werden, weil die entsprechenden Java-Datentypen `Integer` und `String` das Interface `java.lang.Comparable` implementieren (s. auch [JavaAPI]).

  Die nachfolgende Tabelle führt die Auswertungsklassen auf, die für Gleichheits- und Vergleichsterme zuständig sind:

Klasse	**Arithmetische Operation**
`SmallerExpression`	Test, ob der erste Ausdruck ein kleineres Ergebnis liefert als der zweite.
`BiggerExpression`	Test, ob der erste Ausdruck ein größeres Ergebnis liefert als der zweite.
`EqualExpression`	Test, ob beide Ausdrücke gleiche Ergebnisse liefern.

  Wie schon für arithmetische Ausdrücke wurde für Vergleiche die gemeinsame abstrakte Oberklasse `ComparisonExpression` erstellt, die die Verwaltung der beiden Operanden eines Vergleichs übernimmt.

  Obwohl nur Auswertungsklassen für den Größer-Vergleich und den Kleiner-Vergleich erstellt wurden, sind in ODL-Abfragen auch die Operatoren größer-gleich und kleiner-gleich möglich. Dies stellt kein Problem dar, weil ein Vergleich $(a \geq b)$ äquivalent zu $\neg(a < b)$ und $(a \leq b)$ äquivalent zu $\neg(a > b)$ ist, sodass die Auswertungsklassen `SmallerExpression` und `BiggerExpression`, kombiniert mit der Auswertungsklasse für die Negation, auch für die Aus-

[2] Trotz der etwas irreführenden Bennenung ist `EqualExpression` eine Unterklasse von `Term`.

wertung der Operatoren größer-gleich und kleiner-gleich eingesetzt werden können.

Im SableCCGenerator sind folgende Methoden für die Verarbeitung von Gleichheits- und Vergleichstermen zuständig:

Methode	Aufgabe
outAEqualExpression	Erstellung eines Gleichheitsterms.
outABiggerSmallerExpression	Erstellung eines Vergleichsterms.
outABiggerComparisonOperator	Erkennung eines Größer-Operators.
outABiggerOrEqualComparison-Operator	Erkennung eines Größer-Gleich-Operators.
outASmallerComparisonOperator	Erkennung eines Kleiner-Operators.
outASmallerOrEqualComparison-Operator	Erkennung eines Kleiner-Gleich-Operators.

- **Erweiterte Syntax des context-Quantors**
  Für die optionalen Parameter eines context-Quantors – zurzeit ist es nur der Hinweistext, der im Eingabedialog angezeigt wird – musste die ODL-Auswertungsklasse ContextQuantifier, der SableCCGenerator und das Interface MetaType mit allen es implementierenden Klassen angepasst werden. Damit zusammenhängende Änderungen an Query- und GUI-Klassen werden im Abschnitt 5.2 besprochen.

  Wir beginnen mit den Änderungen am SableCCGenerator: hier musste die Methode outAHintExtension eingeführt werden, die den Hinweistext verarbeitet. Außerdem muss nun in der Methode caseAQuantifierUnaryProposition, die für die Verarbeitung von Quantoren zuständig sind, im Codeabschnitt für den context-Quantor die Anwesenheit des optionalen Parameters überprüft werden – ist dieser vorhanden, so wird er an die generierte ContextQuantifier-Instanz weitergegeben.

  Die Klasse ContextQuantifier wurde um die Möglichkeit erweitert, einen Hinweistext optional zu spezifizieren – ist dieser vorhanden, so wird er bei der Auswertung des context-Quantors stets an die query-Methode des Metatyps der einzugebenden Variable weitergegeben, damit er im Eingabedialog dem Benutzer angezeigt werden kann.

  Im Interface MetaType und seinen sämtlichen Unterklassen wurde die Methode query um den Parameter hintText erweitert, über den der Hinweistext spezifiziert wird, der in einem Eingabedialog dem Benutzer angezeigt werden muss.

5.1.3 Weitere Implementierungsaspekte

Wir wollen nun einige weitere implementierungstechnischen Aspekte vorstellen, die sich nicht eindeutig in einem der vorherigen Abschnitte unterbringen ließen und deshalb einen eigenen Abschnitt beanspruchen.

- Term-Hierarchie
  Zunächst wollen wir noch einmal auf den Aufbau der Term-Hierarchie im Package quest.odl.evaluation.model eingehen. Es wurden zwar nicht viele neue Klassen in diese Hierarchie eingefügt, sie erfuhr jedoch einige Umbaumaßnahmen, die vor Allem darauf abzielten, gemeinsame Codeabschnitte aus verwandten Klassen in gemeinsame Oberklassen auszulagern. Folgende Klassengruppen wurden einbezogen:
  - Junktionen
    Die Klassen Conjunction, Disjunction, Equivalence und Implication, die logische Verknüpfungen zwischen zwei Termen implementieren, erhielten eine gemeinsame

abstrakte Oberklasse `Junction`, die die Verwaltung der beiden Operanden der Verknüpfungen übernimmt.

– Quantoren
Für die Quantor-Klassen `ExistentialQuantifier`, `UniversalQuantifier`, `ContextQuantifier` und `NewQuantifier` wurde die gemeinsame abstrakte Oberklasse `Quantifier` erstellt – sie übernimmt die Verwaltung der vom Quantor gebundenen Variablen, ihres Typs und des vom Quantor auszuwertenden Terms.

– Relationen
Für die Klassen `RelationAdditionTerm`, `RelationRemovalTerm` und `RelationTestTerm`, die jeweils für das Hinzufügen, Entfernen oder die Abfrage eines Attributs oder einer Entität zuständig sind, die über eine Relation mit einer anderen Entität verknüpft ist, wurde die gemeinsame abstrakte Oberklasse `RelationTerm` erstellt. Analog zu den oben genannten abstrakten Oberklassen übernimmt sie die Verwaltung der Operanden eines Relationsterms.

Ein vollständiges Klassendiagramm der `Term`-Hierarchie wird im Anhang A auf der Abbildung A.4 gezeigt.

- Package `quest.odl.evaluation.model.analysis`
Für den ODL-Syntaxbaum, der vom SableCC-Parser erstellt wird, generiert SableCC im Package `quest.odl.parser.analysis` das Interface `Analysis` und einige leere Implementierungen dieses Interfaces, die das *Visitor*-Entwurfsmuster für die Syntaxbaum-Knoten implementieren. Dies wurde am Anfang des Abschnitts 5.1 erläutert.

  Nach dem gleichen Schema wird das *Visitor*-Entwurfsmuster für ODL-Auswertungsbäume im Package `quest.odl.evaluation.model.analysis` realisiert. Folgende Klassen bilden das Grundgerüst:

Klasse	**Aufgabe**
`EvalTreeNode`	Interface, das von jedem ODL-Auswertungsbaum-Knoten zu implementieren ist. Dies wird bewerktstelligt, indem die Interfaces `Expression`, `Term` und `MetaType` im Package `quest.odl.evaluation.model` von diesem Interface erben. Da jede ODL-Auswertungsklasse eines dieser Interfaces implementiert, implementiert sie auch das Interface `EvalTreeNode`.
`EvalTreeVisitor`	Inteface eines Besuchers für ODL-Auswertungsbäume: hier wird für jede ODL-Auswertungsklasse `Foo` die Methode `caseFoo( Foo node )` definiert.

In Anlehnung an die Struktur von `quest.odl.parser.analysis` hat das Interface `EvalTreeVisitor` zwei leere Implementierungen:

Klasse	**Aufgabe**
`EvalTreeVisitorAdapter`	Liefert für jede in `EvalTreeVisitor` definierte Methode eine leere Implementierung.
`DepthFirstEvalTreeVisitorAdapter`	Für jede ODL-Auswertungsklasse `Foo` werden die Methoden `inFoo` und `outFoo` definiert, die aus der Methode `caseFoo` aufgerufen werden. Der ODL-Auswertungsbaum wird in der DepthFirst-Reihenfolge traversiert.

Hier muss angemerkt werden, dass das Package `quest.odl.evaluation.model.`

analysis, im Unterschied zu quest.odl.parser.analysis, nicht automatisch erzeugt, sondern von Hand programmiert wurde. Bei Änderungen an ODL-Auswertungsklassen müssen die ggf. notwendigen Änderungen an den Interfaces EvalTreeVisitor und EvalTreeNode sowie an ihren Unterklassen von Hand durchgeführt werden.

Im Package quest.odl.evaluation.model.analysis sind zwei weitere konkrete Implementierungen von EvalTreeVisitor enthalten:

Klasse	**Aufgabe**
TermToStringConverter	Erzeugt für einen ODL-Auswertungsbaum seine String-Darstellung – hierbei handelt es sich um eine ODL-Abfrage, deren Kompilation den analysierten ODL-Term ergeben würde, die aber nicht notwendigerweise dem ursprünglichen ODL-Abfrage-String gleich ist, denn unterschiedlich formulierte ODL-Abfragen können den gleichen ODL-Auswertungsbaum ergeben.
TermVariableFinder	Findet alle innerhalb eines gegebenen ODL-Terms verwendeten und alle in ihm deklarierten Variablen.

Viele Implementierungen von EvalTreeVisitor sind interne Klassen in anderen Klassen. Im SableCCGenerator erben folgende Klassen von DepthFirstEvalTreeVisitorAdapter:

– ContextQuantifierDisplayVariablesSetter wird benutzt, um in einem ODL-Auswertungsbaum bei allen context-Quantoren einzutragen, welche Variablen zum Zeitpunkt der Auswertung dieses Quantors bekannt sind und dem Benutzer im Eingabedialog für die vom Quantor gebundene Variable angezeigt werden müssen.

– ContextQuantifierBackwardStepSetter analysiert einen ODL-Auswertungsbaum und spezifiziert für alle context-Quantoren, ob im Eingabedialog für den betreffenden Quantor ein Rückwärtsschritt möglich ist (s. auch Abschnitt 5.2.3).

Schließlich verwenden folgende Klassen private Unterklassen von EvalTreeVisitorAdapter:

– Klassen im Package quest.odl.evaluation.model.query.dialog:
QueryDialogManager

– Klassen im Package quest.odl.evaluation.model.query.factory:
QueryFactoryManager
ValuesDisplayComponentProducerManager
QueryInputPanelProducerManager
AbstractObjectToStringFormatterFactory

Jede dieser Klassen führt für verschiedene ODL-Datentypen verschiedene Aktionen aus. Die Unterklassen von EvalTreeVisitorAdapter werden hier verwendet, um die Ausführung der richtigen Aktion für eine als Parameter übergebene MetaType-Instanz zu ermöglichen: mit ihrer Hilfe wird festgestellt, um welchen Datentyp es sich bei dieser MetaType-Instanz handelt, und die diesem Datentyp entsprechende Aktion wird ausgeführt.

- Vorziehen von context-Quantoren bei der Skolem-Optimierung
In der früheren ODL-Version wurde die Skolem-Optimierung für new- und context-Quantoren implementiert ([Pasch], S.38-39). Hierbei handelt es sich um das Vorziehen von new- und context-Quantoren bis zum nächsten forall-Quantor oder an den Anfang der

ODL-Abfrage. Das einfache Vorziehen der Quantoren war möglich, weil die Typen der von ihnen gebundenen Variablen von keinen anderen Variablen abhängen konnten. Mit der aktuellen Erweiterung des Typsystems wurden Typen eingeführt, die von früher deklarierten Variablen abhängen können: beispielsweise hängt in der Abfrage

```
exists c1:Component.
  context c2:{ comp:Component | is SubComponents(c1,comp) }
```

der eingeschränkte Typ `{ comp:Component | is SubComponents(c1,comp) }` von der Variablen `c1` ab. In einem solchen Fall kann der betreffende `context`-Quantor nicht vor den Quantor vorgezogen werden, von dessen Variablen sein eigener Variablentyp abhängt. Die `new`-Quantoren sind von dieser Problematik nicht betroffen, da sie nur Variablen des Typs `Entity` binden können, und dieser Typ hängt nie von anderen Variablen ab.

Das entstandene Problem bei der Verschiebung von `context`-Quantoren kann auf verschiedenen Wegen gelöst werden. Die optimale Lösung, bei der ein `context`-Quantor genau soweit vorgezogen wird, bis er auf einen `forall`-Quantor trifft oder einen anderen Quantor, von dessen Variablen sein Variablentyp abhängt, konnte aufgrund ihrer Komplexität nicht im Rahmen der vorliegenden Arbeit implementiert werden. Sie wird jedoch ausführlich im Abschnitt 6.2.1 besprochen.

In der aktuellen Version des ODL-Systems wird eine einfachere jedoch sichere Lösung eingesetzt. Dabei wird für einen `context`-Quantor zunächst ermittelt, ob sein Variablentyp von einer anderen Variablen abhängt. Ist dies nicht der Fall, so wird der Quantor – wie schon früher – bis zum nächsten `forall`-Quantor bzw. an den Anfang der ODL-Abfrage vorgezogen. Hängt der Typ der vom `context`-Quantor gebundenen Variablen von einer anderen Variablen ab, wird der Quantor als unbeweglich markiert und an seiner Stelle belassen. Dieser Algorithmus resultiert zwar häufig in suboptimaler Positionierung von `context`-Quantoren, er ist aber wesentlich einfacher, als die optimale Lösung, und auch mit diesem Algorithmus kann der Benutzer durch Umformulierung der ODL-Abfrage eine optimale Positionierung von `context`-Quantoren erreichen.

5.2 Interaktive Benutzerschnittstelle

Nachdem wir die ODL-Auswertungsklassen besprochen haben, wollen wir eine Beschreibung der im Rahmen dieser Arbeit entwickelten Benutzerschnittstelle für die Eingabe von Variablenwerten geben.

In der früheren ODL-Version wurde der Grundstein für das Query-Subsystem gelegt, das Benutzereingaben ermöglichen soll, die bei der Auswertung von `context`-Quantoren notwendig sind ([Pasch], S.29-30). Die Struktur und Funktionsweise des ODL-Query-Subsystems im Package `quest.odl.evaluation.model.query` wird ausführlich im Abschnitt 5.2.2 besprochen. Vorher wollen wir die GUI-Klassen beschreiben, die für den Aufbau von Benutzereingabedialogen erstellt wurden und von Query-Klassen verwendet werden.

5.2.1 GUI-Klassen

Die GUI-Klassen für das Query-Subsystem belegen zwei Packages:

– `quest.odl.evaluation.model.query.dialog`:
  Dieses Packages enthält die GUI-Klassen, die in Eingabedialogen Verwendung finden.

– `quest.odl.evaluation.model.query.factory`:
  In diesem Package befinden sich Fabrikklassen, die für die Herstellung von GUI-Klassen aus `quest.odl.evaluation.model.query.dialog` zuständig sind – sie implementieren die Entwurfsmuster *Abstrakte Fabrik, Fabrikmethode* und *Erbauer* ([GammaEtAl], S.107-143). Eine weitere Fabrikklasse produziert Instanzen von Klassen aus den Packages `quest.`

dialogs.cellRenderers.formatters und quest.odl.evaluation.model.cellRenderers.formatters.

Zusätzlich zu diesen Klassen wird in Eingabedialogen die Navigationsleiste aus dem Package quest.dialogs.navigationBar verwendet.

Schließlich werden zur Anzeige von Werten verschiedener Datentypen Klassen aus den Packages quest.dialogs.cellRenderers, quest.dialogs.cellRenderers.formatters und aus quest.odl.evaluation.model.cellRenderers.formatters.

Wir wollen nun alle verwendeten Klassengruppen im Einzelnen besprechen.

Package quest.odl.evaluation.model.query.dialog

Die meisten Klassen im dialog-Package lassen sich in drei größere Gruppen unterteilen, die unten aufgeführt werden. In jeder Gruppe werden in der ersten Zeile der Klassenaufzählung abstrakte Klassen aufgeführt, die das Interface der Hierarchie definieren. Ihnen folgen Unterklassen und eventuell weitere benutzte Klassen. Interfaces und abstrakte Klassen werden kursiv gesetzt, wobei Interfaces zusätzlich mit dem Wort *«Interface»* gekennzeichnet werden:

- **Eingabedialog**
  Dialogfenster für Benutzereingaben. Klassen:
  QueryDialog «Interface», *AbstractQueryDialog*,
  DefaultQueryDialog, SplitPanelQueryDialog
  QueryDialogManager

- **Eingabepanel**
  Eingabebereiche für verschiedene ODL-Datentypen, die in Eingabedialogen zur Eingabe des abgefragten Variablenwerts benutzt werden. Für einige Datentypen sind mehrere unterschiedliche Eingabebereiche vorhanden. Klassen:
  QueryInputPanel «Interface», *AbstractQueryInputPanel*,
  TextFieldQueryInputPanel, IntegerTextFieldQueryInputPanel,
  BooleanTextFieldQueryInputPanel,
  SelectionQueryInputPanel «Interface»,
  AbstractSelectionQueryInputPanel,
  ListQueryInputPanel, RadioButtonsQueryInputPanel,
  CompositeQueryInputPanel «Interface»,
  AbstractCompositeQueryInputPanel,
  DefaultCompositeQueryInputPanel,
  SetTypeQueryInputPanel «Interface», *AbstractSetTypeQueryInputPanel*,
  DefaultSetTypeQueryInputPanel
  QueryInputListener «Interface», QueryInputEvent
  Die Eingabepanels für Mengenwerte benötigen eine weitere Gruppe von Klassen, die in diesen Eingabepanels zur Anzeige bereits in die Menge eingefügter Werte dienen:
  SetValueDisplay «Interface», *AbstractSetValueDisplay*,
  ListSetValueDisplay, TableSetValueDisplay

- **Werteanzeige**
  Komponenten, die in Eingabedialogen zur Anzeige der Werte bereits bekannter Variablen verwendet werden. Klassen:
  AbstractValuesDisplay, DefaultValuesDisplay,

`ValuesDisplayComponent` *«Interface»*,
`TextAreaValuesDisplayComponent`, `TableValuesDisplayComponent`

Das Package enthält zwei weitere Klassen, die in keine der obigen Gruppen eingeordnet werden können: `QueryConfigurationDialog` implementiert einen Konfigurationsdialog für das ODL-Query-Subsystem; die Klasse `NavigationEventException` wird in der Dialogflusskontrolle eingesetzt und wird im Abschnitt 5.2.3 besprochen.

Der Aufbau eines Eingabedialogs wurde auf der Abbildung 4.1 gezeigt. Wir wollen noch einmal diesen Aufbau zeigen, jetzt allgerdings mit der Benennung der Klassen, die für die einzelnen Bereiche des Eingabedialogs zuständig sind (Abbildung 5.4).

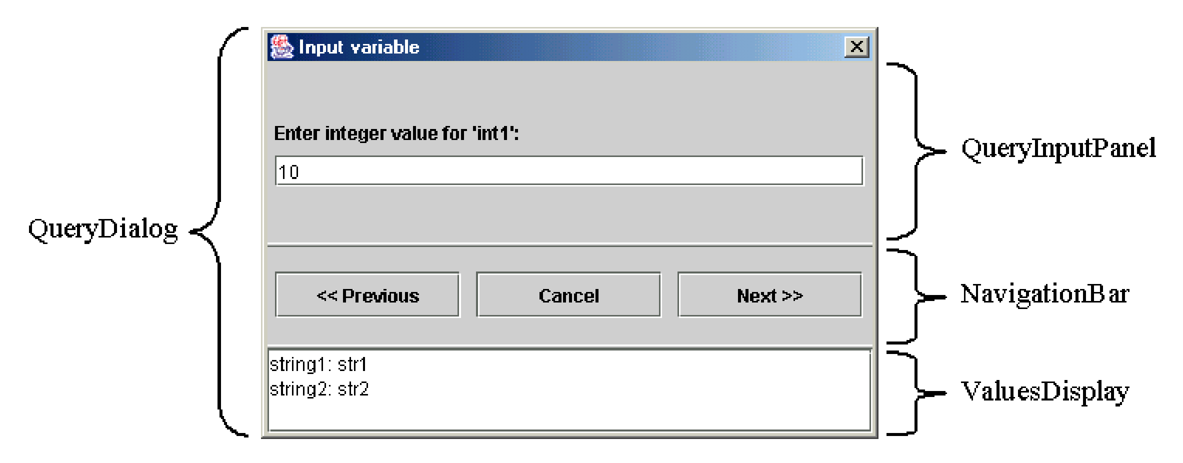


Abbildung 5.4: Aufbau eines Eingabedialogs

Betrachten wir die Implementierung der oben aufgeführten Klassengruppen im Detail.

- **Eingabedialog**
  Das Interface `QueryDialog` definiert die Methoden, die alle Dialogfenster für Benutzereingaben im ODL-Query-Subsystem implementieren müssen – diese Methoden dienen zumeist der Konfiguration der drei Bereiche eines Eingabedialogs (Abb. 5.4). Die Methode `userInput()` startet die Benutzereingabe und wird erst dann verlassen, wenn der Benutzer die Eingabe beendet (durch Bestätigung oder Abbruch).

  Die abstrakte Klasse `AbstractQueryDialog` implementiert die meisten der in `QueryDialog` definierten Methoden und erweitert gleichzeitig die Klasse `javax.swing.JDialog` – dadurch wird festgelegt, dass ein Eingabedialog eine Unterklasse von `JDialog` ist und damit alle Funktionalitäten dieses Standarddialogs zur Verfügung stellt. Um eine konkrete Klasse für ein Dialogfenster zu implementieren, genügt es, von `AbstractQueryDialog` zu erben und das gewünschte Aussehen des Dialogfensters im Konstruktor festzulegen.

  Auf genau diese Weise wurden die beiden konkreten Implementierungen des Eingabedialogs erstellt. `DefaultQueryDialog` ist ein einfacher Eingabedialog (Abb. 4.1 auf S.28), bei dem die Größen der drei Bereiche von einer `java.awt.GridBagLayout`-Instanz automatisch angepasst werden. Die Klasse `SplitPanelQueryDialog` implementiert einen Eingabedialog, bei dem die Größe der drei Bereiche mithilfe von Splittingpanels (`javax.swing.JSplitPane`) vom Benutzer verändert werden kann (Abb. 4.4 auf S.29).

  Um zu vermeiden, dass für jede Benutzereigabe neue Dialogfenster-Instanzen angelegt werden, und um zusätzlich für jeden ODL-Datentyp zur Laufzeit festlegen zu können, welcher Dialogfenstertyp für Benutzereingaben zu verwenden ist, wird je eine Instanz von allen verfügbaren Dialogfenstertypen durch die Klasse `QueryDialogManager` verwaltet. Sie ist ein

Singleton ([GammaEtAl], S.157-166), dessen einzige Instanz von der Methode `getInstance` zurückgeliefert wird. Für die Eingabedialoge `DefaultQueryDialog` und `SplitPanelQueryDialog` stellt der `QueryDialogManager` die Methoden `getDefaultQueryDialogInstance` und `getSplitPanelQueryDialogInstance` bereit, welche die von `QueryDialogManager` verwaltete Instanz der jeweiligen Dialogklasse zurückgeben.

Für jeden ODL-Datentyp (außer dem eingeschränkten Typ, der generell keine eigene Konfiguration für Benutzereingaben benötigt, s. auch Abschnitt 4.2.4) stellt `QueryDialogManager` die Methode `QueryDialog get_ODLType_QueryDialog()` zur Verfügung, die für den jeweiligen Typ die Eingabedialog-Instanz zurückgibt, die für Benutzereingaben zu verwenden ist – hier kommt das *Strategie*-Entwurfsmuster ([GammaEtAl], S.373-384) zur Anwendung. Das Gegenstück zu dieser Methode stellt die Methode `set_ODLType_QueryDialog( QueryDialog dialog )`, die den zu benutzenden Eingabedialog festlegt. Wenn wir beispielsweise den einfachen Eingabedialog für den Typ `String` einstellen wollen, können wir folgenden Code benutzen:

```
QueryDialogManager m = QueryDialogManager.getInstance();
  m.setStringQueryDialog(m.getDefaultQueryDialogInstance());
```

Soll eine eigenständige Dialoginstanz für die Eingaben eines Datentyps benutzt werden, so wird das mit einem Aufruf der Form

```
QueryDialogManager m = QueryDialogManager.getInstance();
  m.setStringQueryDialog(new DefaultQueryDialog());
```

bewerkstelligt. Über die Methode

```
configure( QueryDialog boolDialog, QueryDialog integerDialog,
  QueryDialog stringDialog, QueryDialog entityDialog,
  QueryDialog introducedQueryDialog,
  QueryDialog productTypeDialog, QueryDialog setTypeDialog )
```

kann schließlich mit einem einzigen Aufruf für alle ODL-Datentypen spezifiziert werden, welche Dialogfenster für Benutzereingaben zu verwenden sind.

Ein ausführliches Klassendiagramm der `QueryDialog`-Hierarchie befindet sich im Anhang A auf der Abbildung A.11.

- **Eingabepanel**
  Die abstrakte Klasse `QueryInputPanel` definiert die Schnittstelle von Eingabepanels, die innerhalb eines Eingabedialogs für die Eingabe des Werts der abgefragten Variablen zuständig sind. Von der Idee her handelt es sich bei `QueryInputPanel` um ein Interface, da es aber von `javax.swing.JPanel` erbt, um die Schnittstelle eines `Swing`-Standardpanels zu integrieren, musste es als abstrakte Klasse notiert werden.

  Die wichtigste Methode, die `QueryInputPanel` definiert, ist `Object getInput()`, die den vom Benutzer eingegebenen Wert liefert. Die meisten anderen Methoden sind für das Aussehen des Eingabepanels zuständig.

  Eine weitere Methode, zusammen mit den Klassen `QueryInputEvent` und `QueryInputListener`, implementiert das *Beobachter*-Entwurfsmuster ([GammaEtAl], S.287-300), das im Event-Modell von `AWT` und `Swing` breite Anwendung findet. Die Methode `addQueryInputListener( QueryInputListener listener )` meldet einen Beobachter für Benutzereingabe-Ereignisse bei dem Eingabepanel an. Ein Benutzereingabe-Ereignis, das entweder Änderung der Eingabe (z.B. Auswahl eines anderen Werts in der Auswahlliste) oder Bestätigung der Eingabe (z.B. durch Betätigung des *OK*-Buttons) sein kann, wird durch eine Instanz von `QueryInputEvent` repräsentiert.

Die Klasse AbstractQueryInputPanel implementiert einige Methoden, die in QueryInputPanel definiert wurden. Unter anderem realisiert sie die Verarbeitung von Benutzerereignissen – beim Auftreten eines Benutzerereignisses wird die Methode fireQueryInputEvent(QueryInputEvent evt) aufgerufen, die das Ereignis an alle angemeldeten Beobachter weitergibt.

Die Unterklassen von QueryInputPanel lassen sich in vier Gruppen unterteilen:

- Panels mit textuellem Eingabefeld (z.B. TextFieldQueryInputPanel)
- Panels mit einer Auswahl (z.B. ListQueryInputPanel)
- Kompositum-Panels, die mehrere andere Eingabepanels enthalten können (DefaultCompositeQueryInputPanel)
- Panels für die Eingabe von Mengen (DefaultSetTypeQueryInputPanel)

Wir beschreiben jetzt den Aufbau und die Funktion jeder Gruppe.

Texteingabepanels Die Klasse TextFieldQueryInputPanel realisiert ein einfaches Eingabepanel, bei dem der Variablenwert in einem Textfeld eingegeben werden kann (Abbildung 5.5). Über dem Textfeld befindet sich eine Überschrift, die über die in QueryInputPanel definierte Methode setInputLabel(String label) spezifiziert werden kann.

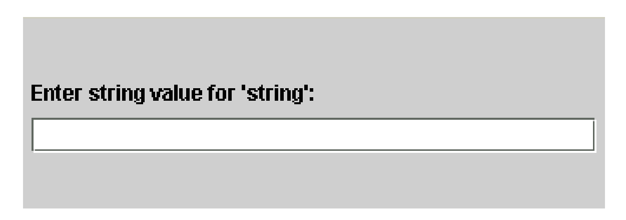


Abbildung 5.5: TextFieldQueryInputPanel

Des Weiteren gehören zu dieser Gruppe die Klassen IntegerTextFieldQueryInputPanel und BooleanTextFieldQueryInputPanel, die von der Klasse TextFieldQueryInputPanel erben und nur die Methode getInput() überschreiben. Der Unterschied zu zwischen diesen beiden Klassen und TextFieldQueryInputPanel besteht darin, dass die Methode getInput() bei diesen Klassen die Eingabe nicht als String-Instanz, sondern als Integer- bzw. Boolean-Instanz zurückgibt – dafür wird der eingegebene Text zu einer ganzen Zahl bzw. zu einem booleschen Wert konvertiert. Diese Eingabepanels ermöglichen es, Int-Werte und Boolean-Werte in einem Textfeld einzugeben.

Auswahlpanels Die nächste Gruppe der Eingabepanels bilden SelectionQueryInputPanel und seine Unterklassen. Sie dienen zur Eingabe von Werten, die aus einer endlichen Wertekollektion ausgewählt werden können.

Die abstrakte Klasse SelectionQueryInputPanel (die wie schon QueryInputPanel von der Idee her ein Interface ist) definiert die Schnittstelle für Eingabepanels, die eine Kollektion von Werten entgegennehmen und diese dann dem Benutzer zur Auswahl anbieten. AbstractSelectionQueryInputPanel implementiert einige Methoden von SelectionQueryInputPanel und stellt die abstrakte Oberklasse für Eingabepanels dar, in denen die Eingabe über die Auswahl eines Werts stattfindet.

Konkrete Implementierungen von SelectionQueryInputPanel sind die folgenden:

- ListQueryInputPanel implementiert die Auswahl über eine Liste, die alle zur Auswahl stehenden Werte aufführt – der Benutzer kann einen von ihnen selektieren (Abb. 5.6).

Zur Anzeige der angebotenen Werte wird eine Instanz von `javax.swing.JList` verwendet, zum Scrollen über die Liste dient eine Instanze von `javax.swing.JScrollPane`.

– `RadioButtonsQueryInputPanel` implementiert die Auswahl über eine Gruppe von Radiobuttons: für jeden Wert wird ein Radiobutton mit der Bezeichnung des Werts erstellt (Abb. 5.7). Es kann höchstens ein Radiobutton zur selben Zeit selektiert werden. Bei der Implementierung wurden `Swing`-Klassen `javax.swing.JRadioButton` und `javax.swing.ButtonGroup` verwendet.

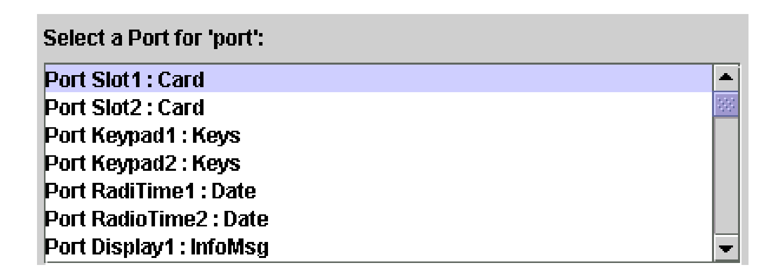


Abbildung 5.6: `ListQueryInputPanel`

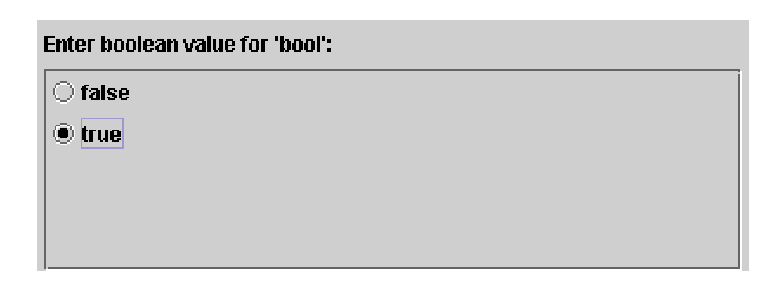


Abbildung 5.7: `RadioButtonsQueryInputPanel`

Zusammengesetzte Eingabepanels Die dritte Gruppe von `QueryInputPanel`-Klassen ist die Gruppe der Kompositum-Eingabepanels. Die abstrakte Ausgangsklasse hier ist `CompositeQueryInputPanel` – sie definiert die Methoden, die das Verhalten von `CompositeQueryInputPanel` als *Kompositum* ([GammaEtAl], S.239-253) für die `QueryInputPanel`-Hierarchie festlegen. So wird über die Methode `setInputPanels( QueryInputPanel[] inputPanels, MetaCompositeType compositeType )` spezifiziert, für welchen zusammengesetzten Typ Werte einzugeben sind und in welchen Eingabepanels es erfolgen soll. Die Methode `CompositeValue getCompositeInput()` gibt den vom Benutzer eingegebenen Wert als `CompositeValue`-Instanz zurück – sie ist die Entsprechung zu der Methode `Object getInput()` in `QueryInputPanel`.

Die abstrakte Klasse `AbstractCompositeQueryInputPanel` implementiert einige der Methoden von `CompositeQueryInputPanel` und stellt die Ausgangsklasse für konkrete Implementierungen eines Eingabepanels für zusammengesetzte Typen dar.

Die Klasse `DefaultCompositeQueryInputPanel` ist eine konkrete Implementierung, die alle Eingabepanels für die einzelnen Elemente des einzugebenden zusammengesetzten Typs nebeneinander anzeigt (Abb. 5.8). Damit der Benutzer die Breite des Eingabebereichs skalieren kann, wird unterhalb des Eingabebereichs ein Slider (`javax.swing.JSlider`) angezeigt, auf dem die geeignete Skalierung gewählt werden kann. Die Möglichkeit, den Slider zu verstecken, wenn er nicht mehr gebraucht wird (Abb. 5.9), wird realisiert, indem der Slider und

der Eingabebereich in einem Splittingpanel (`javax.swing.JSplitPane`) untergebracht werden.

Zwischen `CompositeQueryInputPanel` und `CompositeType` (s. Abschnitt 5.1.2) besteht folgende Verbindung – `QueryInputPanel`-Instanzen geben als Eingabe `Object`-Instanzen zurück; `CompositeQueryInputPanel`-Instanzen liefern `CompositeValue`-Instanzen als Ergebnis der Eingabe zurück: das Kompositum der `QueryInputPanel`-Hierarchie liefert als Benutzereingabe somit Instanzen des Kompositums der Werteklassen-Hierarchie. Hier tritt also noch einmal die Parallelität zwischen dem Aufbau des ODL-Typsystems und des ODL-Query-Subsystems zu Tage.

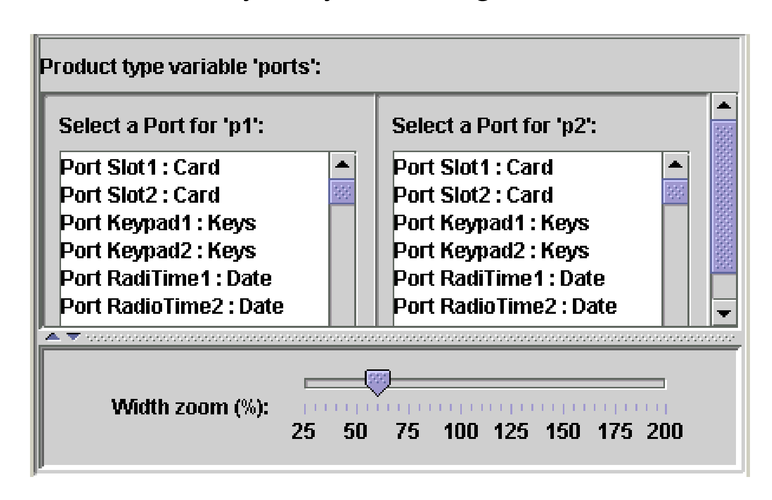


Abbildung 5.8: `DefaultCompositeQueryInputPanel`

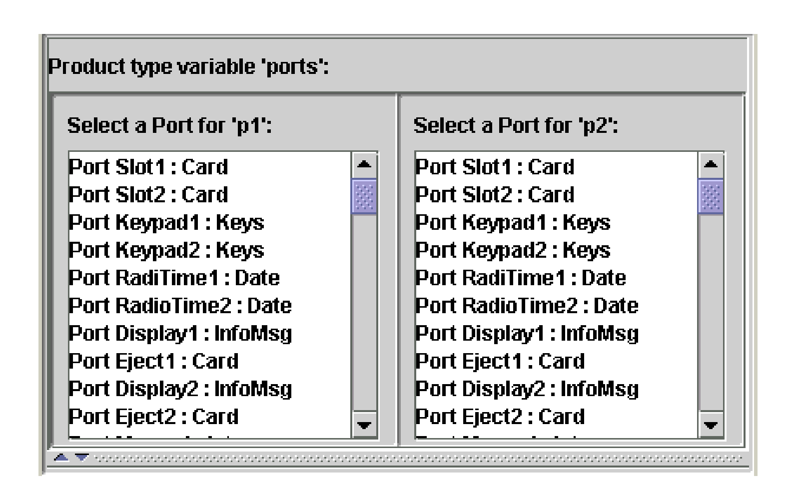


Abbildung 5.9: `DefaultCompositeQueryInputPanel`, Zoomslider minimiert

Mengeneingabepanels Die letzte Klassengruppe in der `QueryInputPanel`-Hierarchie bilden Eingabepanels für Mengen. Die abstrakte Klasse `SetTypeQueryInputPanel` definiert das Interface für diese Klassengruppe. Die wichtigsten Methoden sind `SetType getSetTypeInput()`, welche die vom Benutzer eingegebene Menge als `SetType`-Instanz zurückgibt, und `setInputPanelAndValueDisplay( QueryInputPanel inputPanel, SetValueDisplayvalueDisplay, MetaSetType setType )`, die das Eingabepanel für den Eingabevorgang konfiguriert: der Parameter `setType` spezifiziert den einzugebenden Mengentyp – dieser enthält unter Anderem Angaben über den Basistyp der Menge – und über den Parameter `valueDisplay` wird spezifiziert, welche Komponente zur

Anzeige bereits eingegebener Mengenelemente verwendet werden soll (mehr dazu weiter in diesem Abschnitt).

`AbstractSetTypeQueryInputPanel` implementiert einige Methoden aus `SetTypeQueryInputPanel` und dient als abstrakte Oberklasse für Mengeneingabepanels.

Die Klasse `DefaultSetTypeQueryInputPanel` implementiert ein Eingabepanel für Mengen, das aus zwei Teilen besteht: Eingabebereich für Werte des Basistyps der Menge und Anzeigebereich für Werte, die bereits in die Menge eingefügt wurden (Abb. 5.10). Die beiden Bereiche werden durch die Splittinglinie eines Splittingpanels getrennt, sodass sie nach Bedarf vergrößert und verkleinert werden können.

Wie oben schon erwähnt, verwenden Mengen-Eingabepanels eine zusätzliche Komponente zur Anzeige der bereits in die Menge hinzugefügten Werte. Das Interface `SetValueDisplay` definiert die Schnittstelle einer solchen Komponente. Die Methoden `displayValues( Object[] values )`, `displayValues( Collection values )` und `displayValues( SetValue setValue )` veranlassen die Anzeige der übergebenen Wertemenge – sie unterscheiden sich nur durch den Parametertyp, ansonsten ist ihre Funktionalität identisch. Des Weiteren sind alle Methoden, die das Wort `Selected` oder `Selection` im Methodennamen enthalten, für das Lesen, die Änderung und die Beobachtung der aktuellen Selektion in der Anzeige zuständig: nehmen wir als Beispiel die Methode `int[] getSelectedIndices()`, die die Indizes aller zurzeit selektierten Einträge in der Anzeige zurückgibt, und die Methode `addListSelectionListener( ListSelectionListener listener )`, die einen Beobachter für Selektionsereignisse anmeldet. Die Methode `java.awt.Component getDisplayComponent()` erfüllt eine Funktion, welche an die von `getListCellRendererComponent` im Interface `javax.swing.ListCellRenderer` angelehnt ist – sie gibt eine Komponente zurück, die für die Darstellung der in `displayValues` übergebenen Werte zuständig ist. Bei `SetValueDisplay` handelt es sich also um eine Implementierung des *Strategie*-Entwurfsmusters.

Die abstrakte Klasse `AbstractSetValueDisplay` implementiert viele der in `SetValueDisplay` definierten Methoden und stellt die Oberklasse für Klassen dar, die eine Mengenwert-Anzeige realisieren.

Die Klassen `ListSetValueDisplay` und `TableSetValueDisplay` sind konkrete Implementierungen einer Mengenwert-Anzeige. Die erste Klasse stellt die übergebenen Werte in einer Instanz von `javax.swing.JList` dar (Abb. 5.10), die zweite Klasse nutzt eine `javax.swing.JTable`-Instanz zur Darstellung (Abb. 5.11).

Ein ausführliches Klassendiagramm der `QueryInputPanel`-Hierarchie findet sich sich im Anhang A auf der Abbildung A.10.

- **Werteanzeige**
  Die dritte Klassengruppe im `dialog`-Package ist für die Anzeige der Werte bereits bekannter Variablen in Eingabedialogen verantwortlich.

  Das Interface `ValuesDisplay` definiert Methoden, die für die Anzeige von Variablenwerten benutzt werden: die Methode `displayValues( Collection names, Collection values )` dient zur Anzeige von Variablennamen und Variablenwerten, die sich jeweils in den Kollektionen `names` und `values` befinden (beide Kollektionen müssen von gleicher Größe sein). Die Methode `Component getDisplayComponent()` liefert, wie schon die gleichnamige Methode in der oben behandelten Klasse `SetValueDisplay`, eine Instanz von `java.awt.Component`, die die Variablenwerte darstellt.

  Die abstrakte Klasse `AbstractValuesDisplay` implementiert viele der in `ValuesDisplay` definierten Methoden und dient als Oberklasse für konkrete Implementierungen einer Werteanzeige.

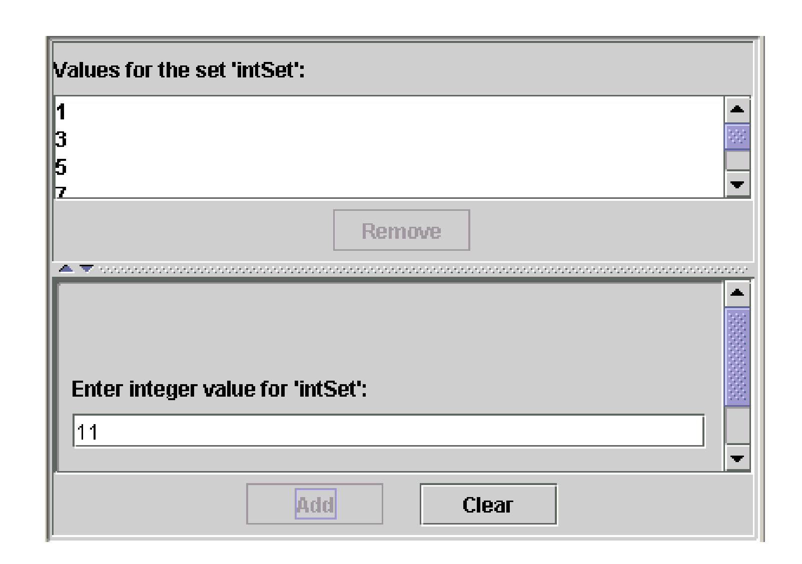


Abbildung 5.10: `SetTypeQueryInputPanel` mit `ListSetValueDisplay`

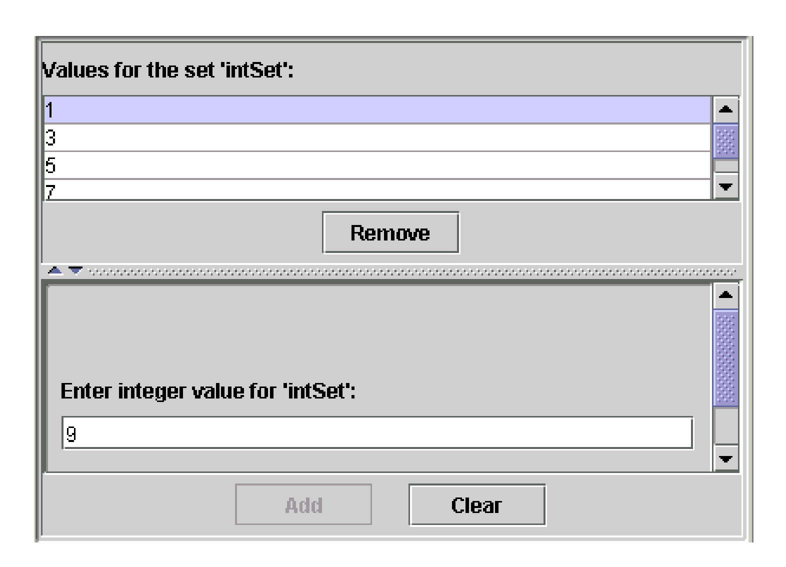


Abbildung 5.11: `SetTypeQueryInputPanel` mit `TableSetValueDisplay`

Die Klasse `DefaultValuesDisplay` liefert eine konkrete Implementierung für eine Werteanzeige. Sie ist allerdings nicht selbst für die Darstellung von Variablenwerten zuständig, sondern bildet eine Indirektionsstufe – die eigentliche Anzeige wird an eine Instanz von `ValuesDisplayComponent` delegiert. Diese Indirektion wird vorgenommen, um die Verwaltung der anzuzeigenden Variablenwerte von ihrer Darstellung zu trennen, und damit verschiedene Darstellungen bei gleichbleibender Verwaltung der Variablenwerte zu ermöglichen. Hierbei handelt es sich somit um eine lokale leichtgewichtige Implementierung des *Model-View-Controller*-Paradigmas ([GammaEtAl], S.5-8) – `DefaultValuesDisplay` spielt hier die Rolle des Modells und `ValuesDisplayComponent` die Rolle des Views (ein Controller ist nicht notwendig, da zurzeit keine Benutzereingaben in Werteanzeigen verarbeitet werden müssen).

Das Interface `ValuesDisplayComponent` definiert lediglich zwei Methoden: `displayValues(Collection names, Collection values)` zeigt die spezifizierten Variablennamen und Variablenwerte an, und `java.awt.Component getDisplayComponent()` liefert eine Komponente, welche die mit dem letzten `displayValues`-Aufruf übergebenen Variablenwerte darstellt.

Konkrete Implementierungen von `ValuesDisplayComponent` sind `TextAreaValuesDisplayComponent` und `TableValuesDisplayComponent`. Die erste Klasse zeigt die Variablennamen und -werte in einem Textbereich an (Abbildung 5.12). Die zweite Klasse benutzt eine zweispaltige Tabelle: in der ersten Spalte werden Variablennamen und in der zweiten die entsprechenden Variablewerte angezeigt (Abbildung 5.13).



Abbildung 5.12: `TextAreaValuesDisplayComponent`

Variable	Value
int1	10
str1	string1
bool1	true

Abbildung 5.13:`TableValuesDisplayComponent`

Ein Klassendiagramm der `ValuesDisplay`-Hierarchie ist auf der Abbildung A.11 im Anhang A zu finden.

- **`QueryConfigurationDialog`**
  Die Klasse `QueryConfigurationDialog` implementiert einen Dialog zur Konfiguration der Benutzerschnittstelle im ODL-Query-Subsystem (s. auch Abschnitt 4.2.7). Sie wurde so konzipiert, dass das Hinzufügen neuer Optionen und Konfigurationsmöglichkeiten sich möglichst einfach gestalten soll.

  Das Dialogfenster enthält für jeden ODL-Datentyp – außer dem eingeschränkten Typ, der keine eigenen Einstellungen benötigt – ein Konfigurationspanel, in dem die Einstellungen für Eingabedialoge des jeweiligen ODL-Datentyps vorgenommen werden können. Jedes der sieben Konfigurationspanels ist eine Instanz der internen Klasse `QueryConfigurationDialog.QueryConfigurationPanel` (für den Mengentyp wird eine Instanz ihrer Unterklasse `SetTypeQueryConfigurationPanel` benutzt). Das Panel bekommt bei der Initialisierung als Parameter die aktuellen Einstellungen für den jeweiligen ODL-Datentyp und zeigt sie an. Vom Benutzer vorgenommene Einstellungen können jederzeit über dafür vorgesehene Methoden von `QueryConfigurationPanel` ausgelesen werden.

  Die zur Verfügung stehenden Optionen für die einzelnen ODL-Datentypen werden in statischen Arrays am Beginn des Klassenquellcodes hinterlegt: dies ermöglicht es, neue Optionen ohne Änderung der Methoden von `QueryConfigurationDialog` hinzuzufügen (dies gilt natürlich nur, solange keine neuen Einstellungekategorien eingeführt werden, wie dies mit der Mengenwert-Anzeige in Eingabepanels für Mengentypen der Fall ist). Auf diese Art und Weise könnte beispielsweise mit minimalem Aufwand ein neues Eingabepanel für einen ODL-Datentyp in die Optionenliste hinzugefügt werden, sodass der Benutzer es später als die zu verwendende Einstellung auswählen kann.

  Eine ausführlicher Kommentar zu `QueryConfigurationDialog` findet sich im Quellcode und in [ODLAPI].

Navigationsleiste

Die Navigationsleiste, die in Eingabedialogen zum Einsatz kommt, wurde im Rahmen eines früheren Projekts erstellt und befindet sich im Package `quest.dialogs.navigationBar`.

Eine ausführliche Beschreibung der Navigationsleiste und dazugehöriger Klassen gibt es in [Tracht] (S.22-25). Wir werden hier daher nur kurz die grundlegenden Funktionen und geringfügige Erweiterungen der Navigationleiste erläutern.

Eine Navigationsleiste wird in Vorgängen verwendet, die in mehreren Schritten ablaufen, um zwischen den einzelnen Schritten zu navigieren. Sie bietet dafür die Buttons *Previous*, *Next* und *Cancel* (Abbildung 5.14), sowie optional einen *Finish*-Button. Jeder Navigationsbutton kann aktiviert oder deaktiviert werden. Bei der Betätigung eines Navigationsbuttons durch den Benutzer wird ein Ereignis (`NavigationBarEvent`) generiert, das an alle angemeldeten Beobachter (`NavigationBarListener`) weitergeleitet wird. Eine Kontrollklasse, die den Ablauf eines Vorgangs steuert, muss sich also als Beobachter bei der Navigationsleiste eintragen. Die Klasse `AbstractQueryDialog` benutzt eine interne anonyme Beobachterklasse, um auf Navigationsereignisse zu reagieren.

Weiterhin bietet eine Navigationsleiste zwei Textbereiche an, in denen ein Hinweistext und eine textuelle Information angezeigt werden können – Eingabedialoge nutzen zurzeit nur den Textbereich für Hinweise (Abbildung 5.15).

Die Klasse `NavigationBar` erfuhr im Rahmen der vorliegenden Arbeit eine geringfügige Erweiterung – die Navigationsbuttons und die Textbereiche können jetzt einzeln versteckt und wieder angezeigt werden: dafür sind neue Methoden zuständig, die das Wort `Visible` im Namen enthalten – beispielsweise wird mit der Methode `setHintVisible( boolean visible )` der Hinweistext angezeigt bzw. versteckt, und die Methode `boolean isHintVisible()` gibt an, ob der Hinweistext sichtbar ist.

Ein Klassendiagramm des Packages `quest.dialogs.navigationBar` befindet sich im Anhang A auf der Abbildung A.13.



Abbildung 5.14: `NavigationBar`

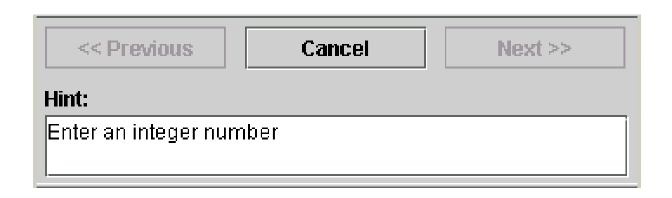


Abbildung 5.15: `NavigationBar` mit Hinweistext

Cellrenderer und Cellrenderer-Formatierer

Um Werte verschiedener Datentypen, vor Allem jedoch Entitäten, angemessen darstellen zu können, genügt die von der Methode `toString()` gelieferte String-Repräsentation oft nicht. Daher wurde für Listen und Tabellen die Möglichkeit genutzt, eigene Cellrenderer zu spezifizieren (s. [JavaAPI], `javax.swing.ListCellRenderer` und `javax.swing.table.TableCellRenderer`), der für die in Listen bzw. Tabellen anzuzeigenden Objekte eine Stringrepräsentation erstellt. Um für ein und dieselbe Klasse, deren Instanzen dargestellt werden sollen, keine zwei Cellrenderer, nämlich eine Implementierung von `ListCellRenderer` und eine von `TableCellRenderer`, erstellen zu müssen, wurde die Implementierung eines Cellrenderers von der Berechnung der Stringdarstellung eines Objekts abgekoppelt.

Zu diesem Zweck wurde das *Strategie*-Entwurfsmuster wie folgt implementiert: das Interface `ObjectToStringFormatter` aus dem Package `quest.dialogs.cellRenderers.`

formatters ist der Ausgangspunkt für alle Klassen, die Stringdarstellungen von Objekten erstellen. Die einzige Methode, die in diesem Interface definiert wird, ist String objectToString(Object object). DefaultObjectToStringFormatter ist eine konkrete Implementierung, die auch als Oberklasse für weitere Formatierer geeignet ist, weil sie einige Methoden implementiert, welche die Realisierung neuer Formatierungen erleichtern. Die Klassen FormattableListCellRenderer und FormattableTableCellRenderer benutzen ObjectToStringFormatter-Instanzen, um die Stringrepräsentation der darzustellenden Objekte zu erhalten. Eine ausführliche Beschreibung dazu findet sich in [Tracht] (S.25-28).

Um Werte verschiedener ODL-Datentypen in Listen und Tabellen darstellen zu können wurden im Package quest.odl.evaluation.model.cellRenderers.formatters folgende Unterklassen von DefaultObjectToStringFormatter erstellt:

- Formatierer für Entitäten
  Für die Entitäten vom Typ Automaton, Channel, Component und Port wurden die entsprechenden Formatierklassen AutomatonEntityToStringFormatter, ChannelEntityToStringFormatter, ComponentEntityToStringFormatter und PortEntityToStringFormatter erstellt. Sie erzeugen Stringrepräsentationen von Entitäten des jeweiligen Typs, die den Namen und eventuelle weitere Informationen über die Entitäten, z.B. den zugewiesenen Datentyp bei Ports.

- Formatierer für zusammengesetzte Typen
  Die Klasse CompositeValueToStringFormatter dient zur Darstellung von zusammengesetzten Werten, indem sie die Stringrepräsentationen der Elemente des zusammengesetzten Werts durch entsprechende ObjectToStringFormatter erzeugt und diese anschließend zu einem String konkateniert. Sie spielt damit die Rolle des *Kompositums* in der ObjectToStringFormatter-Hierarchie.

- Formatierer für Mengentypen
  SetTypeValueToStringFormatter benutzt zur Darstellung einer Menge den als Parameter übergebenen ObjectToStringFormatter für den Basistyp der Menge – dieser berechnet die Stringrepräsentationen aller Mengenelemente, die dann vom SetTypeValueToStringFormatter zu einem String konkateniert werden, der die Menge darstellt.

Die Abbildung A.12 im Anhang A zeigt das Klassendiagramm der im ODL-System verwendeten ObjectToStringFormatter-Klassen.

Package quest.odl.evaluation.model.query.factory

Das factory-Package enthält Fabrikklassen für fast alle Komponenten aus dem Package quest.odl.evaluation.model.query.dialog und einige von den Fabriken verwendete Klassen. Die Fabrikklassen erstellen Instanzen von QueryInputPanel, ValuesDisplayComponent, SetValueDisplay und ObjectToStringFormatter.

Eine Hierarchie von Fabrikklassen beginnt mit einem Interface, das als *Abstrakte Fabrik* agiert und eine *Fabrikmethode* definiert ([GammaEtAl], S.107-118 und 131-143). Die Fabrikmethode wird von konkreten Fabriken implementiert – sie stellen Instanzen der ihnen zugeordneten Klassen aus den Packages quest.odl.evaluation.model.query.dialog und quest.odl.evaluation.model.cellRenderers.formatters her und geben diese zurück.

Für jede Gruppe von Fabrikklassen gibt es eine Manager-Klasse, die mit Fabriken als *Strategie*-Objekten konfiguriert wird und diese anderen Fabriken sowie Query-Klassen aus dem Package quest.odl.evaluation.model.query zur Verfügung stellt.

Die Klassen des factory-Packages können in unten aufgeführte Gruppen unterteilt werden. Interfaces und abstrakte Klassen werden kursiv gesetzt, Interfaces werden zusätzlich mit dem Wort *«Interface»* gekennzeichnet:

- **`QueryInputPanel`-Produzenten**
  In diese Gruppe gehören Klassen, die unkonfigurierte Instanzen von Eingabepanels herstellen – für jede konkrete Implementierung von `QueryInputPanel` gibt es eine entprechende Producer-Klasse, die Instanzen dieses Eingabepanels herstellt. Die Producer-Klassen sind:

  `QueryInputPanelProducer` *«Interface»*,
  `TextFieldQueryInputPanelProducer`,
  `BooleanTextFieldQueryInputPanelProducer`,
  `IntegerTextFieldQueryInputPanelProducer`,

  `SelectionQueryInputPanelProducer` *«Interface»*,
  `AbstractSelectionQueryInputPanelProducer`,
  `ListQueryInputPanelProducer`,
  `RadioButtonsQueryInputPanelProducer`,

  `CompositeQueryInputPanelProducer` *«Interface»*,
  `AbstractCompositeQueryInputPanelProducer`,
  `DefaultCompositeQueryInputPanelProducer`,

  `SetTypeQueryInputPanelProducer` *«Interface»*,
  `AbstractSetTypeQueryInputPanelProducer`,
  `DefaultSetTypeQueryInputPanelProducer`,

  `QueryInputPanelProducerManager`

  Das Interface `QueryInputPanelProducer`, das den Ausgangspunkt der Hierarchie bildet, definiert die *Fabrikmethode* `QueryInputPanel createQueryInputPanelInstance()` – sie gibt `QueryInputPanel`-Instanzen unverändert zurück, so wie sie vom `new`-Befehl erstellt wurden. Nehmen wir als Beispiel `TextFieldQueryInputPanel`: in der entsprechenden Producer-Klasse `TextFieldQueryInputPanelProducer` sieht diese Methode folgendermaßen aus:

```
public QueryInputPanel createQueryInputPanelInstance() {
  return new TextFieldQueryInputPanel();
}
```

  Für die anderen `QueryInputPanel`-Klassen sind entsprechende Producer-Klassen völlig analog implementiert.

  Die *Singleton*-Klasse `QueryInputPanelProducerManager` speichert die Einstellungen darüber, welcher `QueryInputPanel`-Klasse für welchen ODL-Datentyp verwendet werden soll, indem sie für jeden ODL-Datentyp (außer wiederum dem eingeschränkten Typ) die Producer-Klasse speichert, die Eingabepanels für diesen Datentyp herstellt. Dafür sind die Methoden `get_ODLType_QueryInputPanelProducer` und `set_ODLType_QueryInputPanelProducer` zuständig, wobei _ODLType_ für jedes der Wörter `Bool`, `Integer`, `String`, `Entity`, `IntroducedType`, `ProductType` und `SetType` stehen kann.

  Stellt der Benutzer im `QueryConfigurationDialog` beispielsweise für den ODL-Datentyp `Boolean` eine Liste als Eingabepanel ein, so wird im `QueryInputPanelProducerManager` der `ListQueryInputPanelProducer` für diesen Typ eingestellt. Dies geschieht mit den Aufruf:

```
QueryInputPanelProducerManager.getInstance().
  setBoolQueryInputPanelProducer(
    new BooleanTextFieldQueryInputPanelProducer() );
```

Mit der Methode `QueryInputPanelProducer getQueryInputPanelProducer( MetaType type )` wird für einen beliebigen Datentyp zunächst festgestellt, um welchen ODL-Datentyp es sich handelt, und anschließend der entsprechende `QueryInputPanelProducer` zurückgegeben – dafür benutzt `QueryInputPanelProducerManager` eine interne Unterklasse von `EvalTreeVisitorAdapter`.

- **`InputVerifier`-Implementierungen**
  Diese Gruppe enthält Unterklassen von `javax.swing.InputVerifier`. Sie werden von `QueryInputPanel`-Fabriken zur Konfiguration von Eingabepanels verwendet: ein `InputVerifier` überprüft, ob der in einem Eingabepanel eingegebene Werte für den abgefragten ODL-Datentyps zulässig ist. Die Klassen sind:

  - `NotNullInputVerifier`
    Lässt alle Eingaben zu, die ungleich `null` sind.
    → Wird für die Eingabe von `String`-Werten verwendet.
  - `BoolInputVerifier`
    Überprüft, ob die Eingabe ein gültiger boolescher Wert ist, d.h., ob der String, der von der Methode `toString()` der Objekts geliefert wird, welches die Eingabe repräsentiert, gleich ”`true`” oder ”`false`” ist.
    → Wird in Eingabepanels für `Boolean`-Werte eingesetzt.
  - `IntegerInputVerifier`
    Stellt sicher, dass die Eingabe eine gültige ganze Zahl ist – dafür muss der von der Methode `toString()` gelieferte String zu einer ganzen Zahl konvertierbar sein.
    → Wird in Eingabepanels für `Int`-Werte eingesetzt.
  - `CompositePanelInputVerifier`
    Überprüft für einen zusammengesetzten Typ, ob die eingegebenen Werte für alle Elemente des Typs korrekt sind – dies ist der Fall, wenn alle `InputVerifier` der Elemente des Typs ihre jeweiligen Elementwerte akzeptieren.
    → Wird in Eingabepanels für zusammengesetzte Typen verwendet. Das Eingabepanel muss eine Instanz von `CompositeQueryInputPanel` sein.
  - `RestrictedTypeInputVerifier`
    Kontrolliert für einen eingeschränkten Typ, dass der eingegebenen Wert die Restriktionsbedingung des eingeschränkten Typs erfüllen.
    → Kann in allen Eingabepanels verwendet werden, weil der eingeschränkte Typ zur Eingabe auf das Eingabepanel seines Basistyps zurückgreift.
  - `CombinedInputVerifier`
    Kombiniert zwei `InputVerifier`-Instanzen und akzeptiert eine Eingabe genau dann, wenn beide `InputVerifier` diese Eingabe akzeptieren.
    → Wird bei der Eingabe von eingeschränkten Typen verwendet – hierbei wird der `InputVerifier` des Basistyps mit dem `InputVerifier` für die Restriktionsbedingung kombiniert, weil der Wert eines eingeschränkten Typs genau dann korrekt ist, wenn er ein zulässiger Wert des Basistyps ist, der zusätzlich die Restriktionsbedingung erfüllt.

- **`SetValueDisplay`-Fabriken**
  Fabrikklassen aus dieser Gruppe sind für die Herstellung von Mengenwert-Anzeigen für Mengeneingabepanels zuständig. Folgende Klassen gehören zu dieser Gruppe:
  `SetValueDisplayFactory` «Interface»,
  `ListSetValueDisplayFactory`,
  `TableSetValueDisplayFactory`,
  `SetValueDisplayFactoryManager`

Wie schon für QueryInputPanelProducer-Klassen gibt es für jede konkrete Implementierung von SetValueDisplay eine entsprechende Fabrik-Klasse – so werden beispielsweise ListSetValueDisplay-Instanzen von ListSetValueDisplayFactory hergestellt.

Die Manager-Klasse SetValueDisplayFactoryManager speichert die Einstellung, welche Komponente in Eingabepanels für Mengenwerte zur Anzeige bereits in die Menge eingefügter Werte benutzt werden soll. Dafür wird die entsprechende Fabrikklasse an die Methode setSetValueDisplayFactory(SetValueDisplayFactory factory) übergeben. Diese Einstellung kann später über die Methode getSetValueDisplayFactory() ausgelesen werden und wird von der Fabrik-Klasse für Mengen-Eingabepanels benutzt.

- **ObjectToStringFormatter-Fabriken**
  Für die Erzeugung von ObjectToStringFormatter-Instanzen zur Darstellung von Werten verschiedener ODL-Datentypen sind folgende Klassen zuständig:

  ObjectToStringFormatterFactory «Interface»,
  AbstractObjectToStringFormatterFactory,
  DefaultObjectToStringFormatterFactory,
  ObjectToStringFormatterFactoryManager

  Das Inteface ObjectToStringFormatterFactory definiert für jeden ODL-Datentyp eine Methode, die einen Formatierer herstellt, der für Werte dieses Datentyps String-Repräsentationen berechnet.

  Die Klasse ObjectToStringFormatterFactoryManager speichert eine ObjectToStringFormatter-Fabrik, die zur Herstellung von Formatierern für verschiedene ODL-Datentypen verwendet werden soll.

  ObjectToStringFormatter-Fabriken werden von QueryInputPanel-Fabriken und von SetValueDisplay-Fabriken genutzt. Denkbar wäre auch ein zukünftiger Einsatz bei der Konfigurierung von Anzeigekomponenten (ValuesDisplayComponent) für Werteanzeigen in Eingabedialogen.

- **QueryInputPanel-Fabriken**
  Die Klassen in dieser Gruppe stellen QueryInputPanel-Instanzen her und konfigurieren sie. Im Unterschied zu den früher beschriebenen Producer-Klassen sind sie nicht einer QueryInputPanel-Unterklasse, sondern einem ODL-Datentyp zugeordnet, für den sie Eingabepanels herstellen. Dementsprechend hat ihre *Fabrikmethode* als Parameter unter Anderem den ODL-Datentyp, für den das QueryInputPanel zu erstellen und zu konfigurieren ist: QueryInputPanel createQueryInputPanel(MetaType metaType, String variableName, Assignment freeVariables).

  Die primäre Aufgabe einer QueryInputPanel-Fabrik ist die Konfiguration eines Eingabepanels für den dieser Fabrik zugeordneten ODL-Datentyp; die Erstellung einer neuen Eingabepanel-Instanz delegiert sie an eine Producer-Klasse, die als *Strategie*-Objekt vom QueryInputPanelProducerManager zurückgegeben wird. Daher sind QueryInputPanel-Fabriken eher als *Erbauer* zu verstehen ([GammaEtAl], S.119-130).

  Die Klassen sind:

  QueryInputPanelFactory «Interface»,
  BoolQueryInputPanelFactory «Interface»,
  EntityQueryInputPanelFactory «Interface»,
  IntegerQueryInputPanelFactory «Interface»,
  StringQueryInputPanelFactory «Interface»,

IntroducedTypeQueryInputPanelFactory «Interface»,
ProductTypeQueryInputPanelFactory «Interface»,
RestrictedTypeQueryInputPanelFactory «Interface»,
SetTypeQueryInputPanelFactory «Interface»,

AbstractBoolQueryInputPanelFactory,
AbstractEntityQueryInputPanelFactory,
AbstractIntegerQueryInputPanelFactory,
AbstractStringQueryInputPanelFactory,
AbstractIntroducedTypeQueryInputPanelFactory,
AbstractProductTypeQueryInputPanelFactory,
AbstractRestrictedTypeQueryInputPanelFactory,
AbstractSetTypeQueryInputPanelFactory,

`DefaultBoolQueryInputPanelFactory`,
`DefaultEntityQueryInputPanelFactory`,
`DefaultIntegerQueryInputPanelFactory`,
`DefaultStringQueryInputPanelFactory`,
`DefaultIntroducedTypeQueryInputPanelFactory`,
`DefaultProductTypeQueryInputPanelFactory`,
`DefaultRestrictedTypeQueryInputPanelFactory`,
`DefaultSetTypeQueryInputPanelFactory`,

CompositeTypeQueryInputPanelsFactory «Interface»,
`DefaultCompositeTypeQueryInputPanelsFactory`,

`QueryFactoryManager`

Die aufgezählten Fabrikklassen werden von Query-Klassen (Abschnitt 5.2.2) benutzt, um ein Eingabepanel für den abgefragten Datentyp zu erstellen und zu konfigurieren, sodass die Benutzereingabe korrekt durchgeführt werden kann.

- **`ValuesDisplayComponent`-Produzenten**
  Fabriken aus dieser Gruppe stellen Werteanzeige-Komponenten, die von der Klasse `DefaultValuesDisplay` zur Anzeige von Variablenwerten benutzt werden. Die Klassen sind:

  ValuesDisplayComponentProducer «Interface»,

  `TextAreaValuesDisplayComponentProducer`,
  `TableValuesDisplayComponentProducer`,

  `ValuesDisplayComponentProducerManager`

  Die Manager-Klasse `ValuesDisplayComponentProducerManager` speichert für jeden ODL-Datentyp außer dem eingeschränkten Typ die Einstellung, welche Werteanzeige-Komponente in Eingabedialogen für den jeweiligen Datentyp verwendet werden soll. Wenn wir, zum Beispiel, festlegen wollen, dass bei der Eingabe von Entitäten eine Tabelle zur Anzeige von bereits bekannten Variablenwerten benutzt werden soll, müssen wir folgenden Code verwenden:

```
ValuesDisplayComponentProducerManager.getInstance().
  setEntityValuesDisplayComponentProducer(
    new TableValuesDisplayComponentProducer() );
```

  Mit der Methode `getEntityValuesDisplayComponentProducer` kann die eingestellte Fabrik-Klasse von der Query-Klasse für Entitäten ausgelesen und zur Erstellung einer Werteanzeige-Komponente genutzt werden.

Das Klassendiagramm für Producer-Klassen findet sich auf der Abbildung A.15 im Anhang A. Ein Klassendiagramm der `QueryInputPanel`-Fabrikklassen wird auf der Abbildung A.14 gezeigt. Die

Abbildung A.16 zeigt Klassendiagramme für die anderen Klassen aus dem Package `quest.odl.evaluation.model.query.factory`.

5.2.2 Query-Klassen

Die Klassen im Package `quest.odl.evaluation.model.query` bilden die Schnittstelle zwischen dem ODL-Auswertungssystem und dem ODL-Query-Subsystem. Eine schematische Darstellung dieser Beziehung ist auf der Abbildung 5.16 gegeben (Rechtecke mit Auslassungspunkten anstatt der Namen sind eingefügt um zu zeigen, dass nicht alle Systemkomponenten aufgeführt werden).

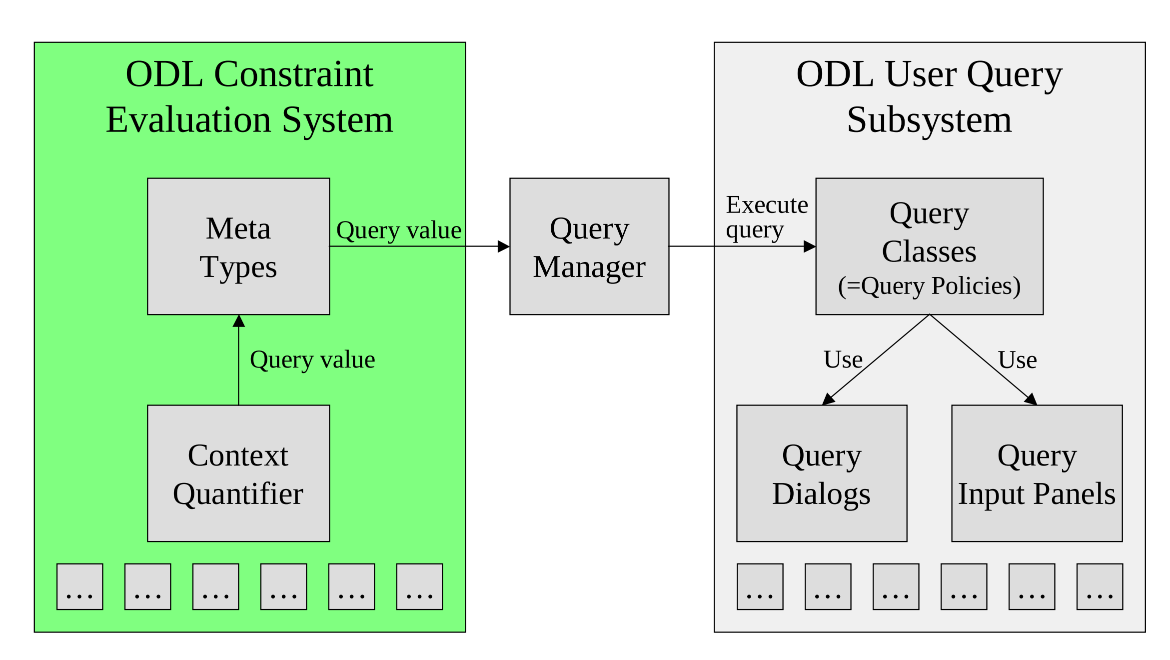


Abbildung 5.16: Schnittstelle zwischen dem ODL-Auswertungssystem und dem ODL-Query-Subsystem

Die Klasse `QueryManager` bildet den Übergangspunkt zwischen dem ODL-Auswertungssystem und dem ODL-Query-Subsystem. Das heißt insbesondere, dass ODL-Auswertungsklassen keine anderen Klassen aus dem Query-Subsystem außer dem `QueryManager` benutzen. Diese modulare Aufteilung ermöglicht es, beliebige Änderungen am Query-Subsystem vorzunehmen – bis hin zur kompletten Ersetzung durch ein anderes Query-Subsystem – ohne das ODL-Auswertungssystem modifizieren zu müssen. Die einzige Bedingung dabei ist, dass das Interface der `QueryManager`-Klasse nicht verändert wird.

Der `QueryManager` ist eine *Singleton*-Klasse, die das *Strategie*-Entwurfsmuster implementiert: für jeden ODL-Datentyp stellt sie eine `query`-Methode zur Verfügung, welche die erforderliche Benutzereingabe von einer dem ODL-Datentyp entsprechenden Query-Klasse ausführen lässt. Die Tabelle 5.13 führt für jeden ODL-Datentyp die entsprechende `query`-Methode der `QueryManager`-Klasse und den Namen des Interfaces, das von Query-Klassen implementiert werden muss, die Benutzereingaben für diesen Datentyp realisieren (diese Klassen dienen als *Strategie*-Komponenten für den `QueryManager` und werden von seinen `query`-Methoden genutzt). Die Parameterlisten der `query`-Methoden wurden aufgrund ihrer Länge und Ähnlichkeit weggelassen.
Bei der Initialisierung wird der `QueryManager` mit Query-Klassen als *Strategie*-Objekten konfiguriert, indem die `configure`-Methode mit den zu benutzenden Query-Klassen als Parametern aufgerufen wird. In der aktuellen ODL-Version findet dieser Aufruf im statischen Initialisierungsblock der Klasse `quest.odl.editor.gui.EditorDialog` statt:

```
QueryManager.instance().configure(
  new DefaultBoolQuery(),
```

ODL-Datentyp	**`query`-Methode im `QueryManager`**	**Interface für Query-Klasse**
`Boolean`	`Boolean queryBool(..)`	`BoolQuery`
`Integer`	`Integer queryInteger(..)`	`IntegerQuery`
`String`	`String queryString(..)`	`StringQuery`
`Entity`	`Entity queryEntity(..)`	`EntityQuery`
`IntroducedType`	`Object queryIntroducedType(..)`	`IntroducedTypeQuery`
`ProductType`	`ProductValue queryProductType(..)`	`ProductTypeQuery`
`RestrictedType`	`Object queryRestrictedType(..)`	`RestrictedTypeQuery`
`SetType`	`SetValue querySetType(..)`	`SetTypeQuery`

Tabelle 5.13: `query`-Methoden und Query-Klassen für verschiedene ODL-Datentypen

```
    new DefaultIntegerQuery(),
    new DefaultStringQuery(),
    new DefaultEntityQuery(),
    new DefaultIntroducedTypeQuery(),
    new CompositePanelProductTypeQuery(),
    new DefaultRestrictedTypeQuery(),
    new DefaultSetTypeQuery() );
```

Eine Benutzereingabe wird bei der Auswertungs eines `context`-Quantors angestoßen. Dieser ruft in der `MetaType`-Instanz, die den Datentyp des einzugebenden Werts beschreibt, die Methode `query` auf. Diese Methode wendet sich ihrerseits an die dem Datentyp entsprechende `query`-Methode in der Klasse `QueryManager`. An dieser Stelle wird die Kontrolle an das Query-Subsystem übergeben. Die `query`-Methode des `QueryManager`'s benutzt die Query-Klasse, die als *Strategie* für Benutzereingaben für den abgefragten Datentyp eingestellt ist, indem sie ihre `query`-Methode aufruft. Diese führt die Benutzereingabe aus und gibt das Eingabeergebnis zurück. Die Abbildung 5.17 zeigt den Ablauf einer Benutzereingabe für einen booleschen Wert (das Sequenzdiagramm wurde aus Gründen der Übersichtlichkeit um wenige Methodenaufrufe gekürzt, die für das Verständnis des Ablaufs keine wesentliche Rolle spielen).

Den Quellcode der `query`-Methode der dafür vom `QueryManager` benutzten Query-Klasse `DefaultBoolQuery` wollen wir im Folgenden kommentiert angeben:

```
public Boolean bQuery(String variableName, Assignment freeVariables,
                     Collection names, Collection values,
                     boolean enableBackwardStep, String hintText ) {
  * Instanz des Eingabedialogs holen, der für boolesche Werte
     benutzt werden soll. */
  QueryDialog queryDialog = QueryDialogManager.getInstance().
    getBoolQueryDialog();

  /* Eingabepanel-Fabrik für den booleschen Datentyp vom
     Eingabepanel-Manager holen. */
  BoolQueryInputPanelFactory inputPanelFactory =
    QueryFactoryManager.getInstance().getBoolQueryInputPanelFactory();

  /* Eingabepanel von der Eingabepanel-Fabrik erzeugen lassen. */
  QueryInputPanel queryInputPanel =
    inputPanelFactory.createBoolQueryInputPanel( variableName );

  /* Das erzeugte Eingabepanel an den Eingabedialog übergeben. */
  queryDialog.setQueryInputPanel( queryInputPanel );

  /* Wenn Variablenwerte übergeben wurden, die dem Benutzer anzuzeigen
     sind, dann eine Werteanzeige erstellen und an den Eingabedialog
     übergeben. */
```

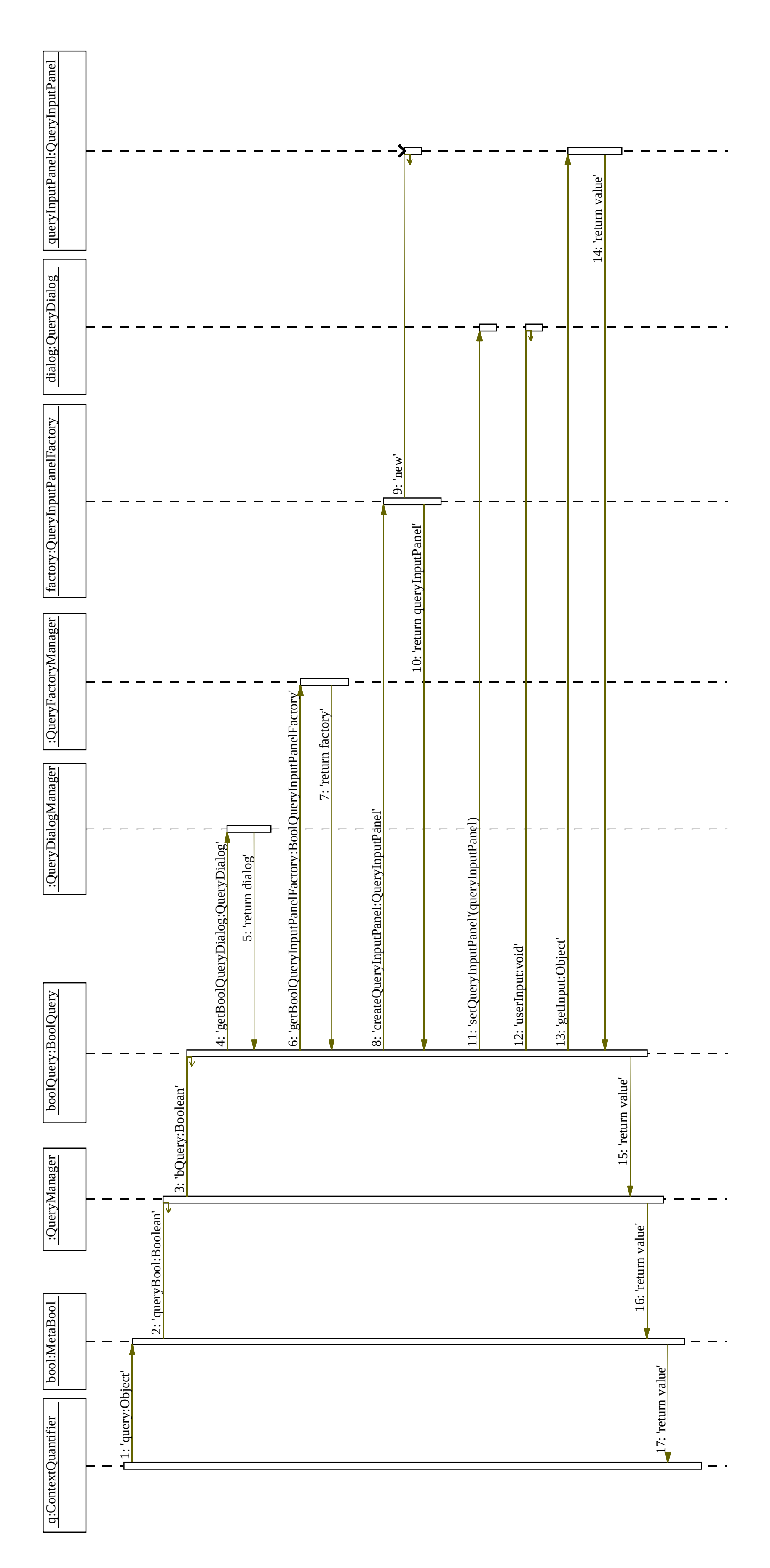


Abbildung 5.17: Sequenzdiagramm einer Benutzereingabe für den Typ `Boolean`

```
    if ( names != null && values != null ) {
     /* Werteanzeige-Producer für Boolean-Abfragen von der
        Manager-Klasse holen. */
      ValuesDisplayComponentProducer valuesDisplayComponentProducer =
        ValuesDisplayComponentProducerManager.getInstance().
          getBoolValuesDisplayComponentProducer();
      /* Werteanzeige erstellen, sie mit der vom Producer erzeugten
         Werteanzeige-Komponente konfigurieren und an den
         Eingabedialog übergeben. */
      queryDialog.setValuesDisplay( new DefaultValuesDisplay(
        valuesDisplayComponentProducer.
          createValuesDisplayComponentInstance() ) );
      /* Variablennamen und -werte in der Werteanzeige darstellen
         lassen. */
      queryDialog.displayValues( names, values );
    } else {
      /* Falls keine Variablenwerte anzuzeigen sind, dann keine
         Werteanzeige übergeben. */
      queryDialog.setValuesDisplay( null );
    }

    /* Festlegen, ob ein Rückwärtsschritt möglich ist. Dies ist der Fall,
       wenn vor der aktuell einzugebenden Variablen bereits andere
       eingegeben wurden: dann kann der Benutzer zur Eingabe der
       vorherigen Variablen zurückkehren. */
    queryDialog.setBackwardStepEnabled( enableBackwardStep );
    /* Den eventuell übergebenen Hinweistext im Eingabedialog anzeigen
       lassen. */
    queryDialog.setHintText( hintText );
    /* Benutzereingabe starten. Diese Methode wird erst dann beendet,
       wenn der Benutzer die Eingabe abgeschlossen hat (wird die Eingabe
       vom Benutzer abgebrochen, so erzeugt dies eine Exception, die die
       query-Methode sofort abbricht. */
    queryDialog.userInput();
    /* Den Eingegebenen Wert aus dem Eingabepanel auslesen
       und zurückgeben. */
    return (Boolean)queryDialog.getQueryInputPanel().getInput();
  }
```

Wie man am Beispiel der oben aufgeführten `query`-Methode sehen kann, spielen Query-Klassen für den Eingabedialog die Rolle des *Erbauers* (s. [GammaEtAl], S.119-130) – es ist die Aufgabe der `query`-Methode einer Query-Klasse, alle Komponenten für den Eingabedialog zu erstellen und ihn für die Eingabe eines Werts des Datentyps zu konfigurieren, für den die jeweilige Query-Klasse zuständig ist.

Ein Klassendiagramm der Query-Klassen befindet sich auf der Abbildung A.9 im Anhang A.

5.2.3 Dialogflusskontrolle

Wie bereits im Abschnitt 4.2.1 beschrieben, kann der Benutzer zwischen den Eingabedialogen für verschiedene Variablen navigieren – aus jedem Eingabedialog kann er durch Betätigen des *Previous*-Buttons zur Eingabe der vorherigen Variablen zurückkehren, und mit dem *Cancel*-Button kann er die gesamte Auswertung einer ODL-Abfrage abbrechen.

Die Dialogflusskontrolle wurde mithilfe des `Exception`-Mechanismus von Java implementiert

– bei einem Rückwärtsschritt oder einem Abbruch im Eingabedialog wird eine `Exception` ausgelöst, die die aktuelle Eingabe unterbricht und die Kontrolle an den Aufrufer zurückgibt. Dieser entscheidet anhand der im `Exception`-Objekt enhaltenen Informationen über das eingetretene Navigationsereignis, welche Aktionen auszuführen sind.

Um den Navigationsmechanismus zu implementieren, wurden die Klassen `UserBreakException` (Package `quest.odl.evaluation`) und `NavigationEventException` (Package `quest.odl.evaluation.model.query.dialog`) erstellt, sowie Modifikationen an den Klassen `ContextQuantifier` und `AbstractQueryDialog` vorgenommen. Außerdem wurde der ODL-Editor (Klasse `quest.odl.editor.gui.EditorDialog`) so angepasst, dass eine im Laufe der ODL-Abfrageauswertung aufgetretene `UserBreakException` abgefangen und die mit ihr übergebene Meldung in der Statuszeile des ODL-Editors angezeigt wird.

Wir wollen die Änderungen bei bestehenden Klassen und die Aufgaben der neuen Klassen erläutern:

– `NavigationEventException`
  Wird von einem Eingabedialog erzeugt, wenn der Benutzer den *Previous*-Button oder den *Cancel*-Button betätigt.

– `AbstractQueryDialog`
  Wurde so erweitert, dass beim Betätigen des *Previous*-Buttons und des *Cancel*-Buttons eine `NavigationEventException` erzeugt wird, die die Information darüber enthält, ob der Benutzer einen Rückwärtsschritt ausführen oder die Auswertung abbrechen will. In beiden Fällen wird der Eingabedialog geschlossen und die erzeugte `Exception` abgesetzt – diese wird vom `ContextQuantifier` abgefangen.

– `UserBreakException`
  Wird von einem `ContextQuantifier` abgesetzt, wenn er eine `NavigationEventException` abfängt, die erzeugt wurde, als der Benutzer den *Cancel*-Button im Eingabedialog betätigte. Eine `UserBreakException` bricht die Auswertung der ODL-Abfrage ab, und wird vom `EditorDialog` abgefangen, der die Meldung aus dem `Exception`-Objekt in der Statuszeile anzeigt.

– `ContextQuantifier.BackwardStepException`
  Interne `Exception`-Klasse von `ContextQuantifier`. Wenn der `context`-Quantor eine `NavigationEventException` abfängt, die durch die Betätigung des *Previous*-Buttons im Eingabedialog abgesetzt wurde, so wird eine `BackwardStepException` erzeugt, die die Auswertung des aktuellen `context`-Quantors abbricht und vom vorherigen `context`-Quantor abgefangen wird – auf diese Art wird ein Rückwärtsschritt ausgeführt.

– `ContextQuantifier`
  Die `evaluate`-Methode wurde so erweitert, dass Navigationsereignisse aus dem Eingabedialog verarbeitet werden können. Die `NavigationEventException`'s werden abgefangen und, in Abhängigkeit davon ob es sich um einen Rückwärtsschritt oder einen Abbruch handelt, verschiedene Aktionen durchgeführt.

Wir wollen uns nun die Implementierung der Dialogflusskontrolle in der `ContextQuantifier`-Klasse näher ansehen. Das folgende Code-Stück enthält eine kommentierte und um unwesentliche Details gekürzte `evaluate`-Methode der `ContextQuantifier`-Klasse.

```
public class ContextQuantifier extends Quantifier implements Term {
  ...
  ...
  public TermResult evaluate(Assignment freeVariables,
                             boolean wantedValue) {
    ...
    ...
    /* Endlosschleife, die verlassen wird, wenn das Ergebnis der
```

```
   Auswertung zurückzugeben ist, oder wenn ein Navigationsereignis
   aufgetreten ist. */
while ( true ) {
  try {
    // Benutzerabfrage für den Variablenwert durchführen lassen
    Object value = variableType.query( ... );

    /* Lokale Kopie der Variablenbelegungen anlegen: sollte der
       Benutzer später mit einem Rückwärtsschritt zu diesem
       context-Quantor zurückkehren, müssen die ursprünglichen
       Variablenbelegungen verfügbar sein. */
    Assignment localFreeVariables = (Assignment)freeVariables.clone();
    /* Vom Benutzer eingegebenen Variablenwert zu den
       Variablenbelegungen hinzufügen. */
    localFreeVariables.put( variableIdentifier, value );
    // Den vom context-Quantor gebundenen Term auswerten.
    TermResult opResult = (TermResult)operandTerm.evaluate(
      localFreeVariables, wantedValue );
    ...

    // Das Ergebnis der Termauswertung zurückgeben
    return opResult;

  } catch ( NavigationEventException navigationEvent ) {
    /* Navigationsexception abgefangen, die im Laufe der
       Benutzereingabe oder bei der Auswertungs des Quantorterms
       aufgetreten ist. */

    if ( navigationEvent.getEventType() ==
         NavigationEventException.INPUT_CANCELLED ) {
      /* Bei einem Abbruch wird die UserBreakException abgesetzt.
         Diese Exception wird nicht mehr von einem anderen
         context-Quantor abgefangen, sondern bis zum ODL-Editor
         weitergereicht, der dann die Meldung
         "User cancelled querying the variable ..." anzeigt.
         Die Auswertung der ODL-Abfrage wird damit abgebrochen.*/
      throw new UserBreakException(
        "User cancelled querying the variable '" +
        variableIdentifier + "'" );

    } else if ( navigationEvent.getEventType() ==
                NavigationEventException.BACKWARD_STEP ) {
      /* Für einen Rückwärtsschritt wird eine BackwardStepException
         abgesetzt: sie wird vom vorhergehenden context-Quantor
         im Auswertungsbaum abgefangen - dieser wird dann die Eingabe
         seiner Variablen wiederholen und die Auswertungs seines
         Quantorterm wieder starten. */
      throw new ContextQuantifier.BackwardStepException();
    }
  } catch ( ContextQuantifier.BackwardStepException e ) {
    /* Eine BackwardStepException wurde abgefangen, die von einem
       nachfolgenden Quantor im Auswertungsbaum erzeugt wurde.
       In diesem Fall muss die Eingabe der Variablen wiederholt
       und die Auswertung des Quantorterms mit dem neuen Wert
       wieder gestartet werden. Dies wird durch die
       "while( true )"-Schleife gewährleistet, die - da hier keine
       Exception geworfen und kein Wert zurückgegeben wird - einfach
       für die Wiederholung des try-Blocks sorgt. */
  }
}
```

```
  }
  ...
  ...
  /** Interne Klasse BackwardStepException. Diese Exception wird von
   *  einem context-Quantor erzeugt, wenn ein Rückwärtsschritt
   *  durchgeführt werden muss: die Auswertung des aktuellen
   *  context-Quantors wird unterbrochen und die Exception wird vom
   *  vorhergehenden context-Quantor abgefangen. */
  private class BackwardStepException extends java.lang.RuntimeException {}
}
```

Im Abschnitt 6.4 werden Ideen für eine flexiblere Dialogflusskontrolle vorgestellt.

5.3 Vorbereitung weitergehender Änderungen

Nachdem wir die Implementierung der neuen Sprachkonstrukte und der interaktiven Benutzerschnittstelle für das ODL-System beschrieben haben, wollen wir einige Hinweise für Weiterentwicklungen geben.

Der allgemeine Aufbau des ODL-Auswertungssystems und die Schritte, die zur Implementierung einer Änderung des ODL-Sprachumfangs notwendig sind, wurden am Beginn des Abschnitts 5.1 bereits besprochen, sodass wir hier auf eine allgemeine Beschreibung verzichten und stattdessen einige Beispiele für konkrete Implementierungen von ODL-Sprachkonstrukten sowie von GUI-Klassen für die Benutzerschnittstelle vorstellen, die im Rahmen dieser Arbeit erstellt wurden.

- **Einführung eines neuen ODL-Ausdrucks am Beispiel von `SetSizeExpression`**
  Wir bereits im Abschnitt 5.1 beschrieben muss zur Einführung eines neuen ODL-Sprachkonstrukts die ODL-Grammatik angepasst, eine ODL-Auswertungsklasse für dieses Konstrukt erstellt und der `SableCCGenerator` um Methoden für die Verarbeitung des Konstrukts ergänzt werden. Wir zeigen diese Schritte anhand des ODL-Ausdrucks für Ermittlung der Größe einer Menge.

  Ausgehend von der Syntax `size( setExpression )` wird die Produktionsregel für den Ausdruck sowie die Deklaration des Tokens `size` in die ODL-Grammatik eingefügt:

  ```
  Tokens
    size = 'size';
  Productions
    arithmetic_term =
      .../* Frühere Ableitungen von arithmetic_term */
      set_size size l_par expression r_par;
  ```

  Als Nächstes wird eine Auswertungsklasse für den neuen Ausdruck erstellt, die das Interface `quest.odl.evaluation.model.Expression` implementiert und in der `evaluate`-Methode die geforderte Auswertung durchführt:

  ```
  public class SetSizeExpression implements Expression {
    /** Ausdruck, der auf der Basis der übergebenen
        Variablenbelegungen die Menge berechnet. */
    Expression setExpression;

    public SetSizeExpression(Expression setExpression)
          throws IllegalArgumentException {
      this.setExpression = setExpression;
    }
  ```

```
    /** Die Menge aus den übergebenen Variablenbelegungen berechnen
        und ihre Größe zurückgeben. */
    public Object evaluate(Assignment freeVariables) {
      SetValue set =
        (SetValue)setExpression.evaluate( freeVariables );
      return new Integer( set.getSize() );
    }

    /** Methode aus dem Interface EvalTreeNode */
    public void apply(EvalTreeVisitor visitor) {
      visitor.caseSetSizeExpression( this );
    }
  }
```

Die Methode `apply` wird im Interface `quest.odl.evaluation.model.analysis.EvalTreeNode` definiert, die zur Implementierung des *Visitor*-Entwurfsmusters für ODL-Auswertungsbäume gehört (mehr dazu im Abschnitt 5.1.3). Als Bestandteil des *Visitor*-Entwurfsmusters wird die `apply`-Methode durch jede ODL-Auswertungsklasse implementiert – hier wird dann die der jeweiligen Klasse entsprechende Methode von `EvalTreeVisitor` aufgerufen. Das bedeutet unter Anderem, dass mit der Einführung einer neuen Auswertungsklasse auch eine entsprechende Methode in `EvalTreeVisitor` eingefügt und die Klassen `EvalTreeVisitorAdapter` und `DepthFirstEvalTreeVisitorAdapter` angepasst werden müssen. Bei der Einführung der Auswertungsklasse `SetSizeExpression` sind demnach folgende Änderungen an diesen Klassen vorzunehmen:

– Interface `EvalTreeVisitor`:
  Hier muss die Methodendeklaration

  ```
  public void caseSetSizeExpression(SetSizeExpression expression);
  ```

  eingefügt werden.

– Klasse `EvalTreeVisitorAdapter`:
  Defaultimplementierung der Methode `caseSetSizeExpression` einfügen:

  ```
  public void caseSetSizeExpression(SetSizeExpression expression){
    defaultCase( expression );
  }
  ```

– Klasse `DepthFirstEvalTreeVisitorAdapter`:
  Die Methode `caseSetSizeExpression` ist zu implementieren, wobei die Struktur der Klasse `SetSizeExpression` berücksichtigt werden muss, d.h., dass für alle Auswertungsklassen, die von `SetSizeExpression` benutzt werden, die Methode `apply` aufzurufen ist:

  ```
  public void caseSetSizeExpression(SetSizeExpression expression){
    inSetSizeExpression( expression );
    expression.getExpression().apply( this );
    outSetSizeExpression( expression );
  }

  public void inSetSizeExpression(SetSizeExpression expression){
    defaultIn( expression );
  }

  public void outSetSizeExpression(SetSizeExpression expression){
    defaultOut( expression );
  }
  ```

Weitere Visitor-Klassen für ODL-Auswertungsbäume, beispielsweise `TermToStringConverter`, sollten ebenfalls um Methoden für die neue Auswertungsklasse ergänzt werden.

Der letzte Schritt ist die Erstellung von Methoden im `SableCCGenerator`, die für die Kompilation des neuen Sprachkonstrukts zuständig sind. Diese Methoden müssen, wenn sie in einem vom SableCC-Parser erstellten ODL-Syntaxbaum einer ODL-Abfrage auf das neue Sprachkonstrukt treffen, eine Instanz der entsprechenden ODl-Auswertungsklasse in den ODL-Auswertungsbaum einfügen:

```
public void outASetSizeArithmeticTerm(ASetSizeArithmeticTerm node) {
  TypedExpression typedExpression =
    (TypedExpression)getOut( node.getExpression() );
  /* Überprüfen, ob der Typ des Ausdrucks, auf den SetSizeExpression
     angewandt wird, ein Mengentyp ist. */
  if ( !( typedExpression.getType().getInstanceMetaType()
          instanceof MetaSetType ) ) {
    throw new InvalidArgumentsException(
      "Argument must be a set",
      typedExpression.getToken().getLine(),
      typedExpression.getToken().getPos() );
  }
  /* Interne Repräsentation für die Auswertungsklasse
     SetSizeExpression erstellen. */
  TypedExpression sizeExpression = new TypedSetSizeExpression(
    node.getSize(), typedExpression );
  setOut( node, sizeExpression );
}
```

Bei `TypedSetSizeExpression` handelt es sich um eine interne Klasse von `SableCCGenerator`, die zusätzlich zu den Informationen für den Aufbau einer `SetSizeExpression` weitere Information enthält, die vom `SableCCGenerator` zur Kompilation einer ODL-Abfrage benötigt werden: ihre Instanz wird im späteren Verlauf der Kompilation der ODL-Abfrage durch eine Instanz von `SetSizeExpression` ersetzt.

Nach der Durchführung aller oben beschriebenen Schritte ist die Einführung des neuen ODL-Ausdrucks abgeschlossen: er kann nun in ODL-Abfragen verwendet werden.

- **Einführung eines neuen ODL-Datentyps am Beispiel von `MetaProductType`:**
  Die ersten drei Schritte für die Einführung eines neuen ODL-Datentyps sind dieselben wie für einen neuen ODL-Ausdruck: Einfügen der Produktionsregeln in die ODL-Grammatik, Erstellung einer entsprechenden ODL-Auswertungsklasse, die diesmal das Interface `MetaType` implementiert, Anpassung von `EvalTreeVisitor` und seiner Unterklassen, Ergänzung von `SableCCGenerator` um Methoden für die Verarbeitung des neuen Typs.

  Der nächste Schritt, der bei einem neuen Datentyp hinzukommt, ist die Implementierung der Benutzereingabe für diesen Datentyp. Ein Datentyp wendet sich an das ODL-Query-Subsystem, indem aus seiner `query`-Methode heraus die dem Datentyp entsprechende Methode der `QueryManager`-Klasse aufgerufen wird – für den Produkttyp heißt sie `queryProductType` und liefert als Rückgabewert eine `ProductValue`-Instanz, die den vom Benutzer eingegebenen Wert enthält. Nachdem das ODL-Query-Subsystem ausführlich im Abschnitt 5.2 beschrieben wurde, verzichten wir hier auf eine ausführliche Beschreibung der Implementierung von Benutzereingaben und geben nur die Klassen an, die bei der Einführung von Produkttypen im ODL-Query-Subsystem erstellt werden mussten:

– `ProductTypeQuery`, `CompositePanelProductTypeQuery`:
  Interface und konkrete Implementierung einer Query-Klasse für Produkttypen, die als *Strategie*-Objekte vom `QueryManager` verwendet werden, um Benutzereingaben auszuführen.
– `CompositeQueryInputPanel`, `DefaultCompositeQueryInputPanel`
  Interface und Implementierung eines Eingabepanels für Produkttypen. Die Erstellung neuer Eingabepanels für einen Typ ist nur dann notwendig, wenn keines der bereits vorhandenen Eingabepanels die Eingabe von Werten dieses Typs übernehmen kann: für Produkttypen ist das der Fall.
– `ProductTypeQueryInputPanelFactory`,
  `DefaultProductTypeQueryInputPanelFactory`
  Interface und Implementierung einer Fabrik für die Erstellung und Konfiguration von Eingabepanels für die Eingabe von Produkttypen.
– `CompositeQueryInputPanelProducer`
  Producer-Klasse für die Erstellung von Eingabepanels für Produktypen, die von `DefaultProductTypeQueryInputPanelFactory` benutzt wird.
– `CompositePanelInputVerifier`
  Ein `InputVerifier` für `CompositeQueryInputPanel`-Instanzen, der überprüft, ob der vom Benutzer eingegebene Wert ein zulässiger Produktwert ist, d.h., ob für alle Elemente des Produkttyps ein zulässiger Wert eingegeben wurde.

Neben der Erstellung neuer Klassen für die Eingabe von Werten des neuen Typs müssen folgende Klassen ergänzt werden:

– `QueryManager`
  Für den neuen Typ wird die `queryProductType`-Methode eingefügt, welche die Benutzereingabe für einen Produkttyp startet.
– `QueryDialogManager`
  Hier müssen die Konfigurationsmethoden `getProductTypeQueryDialog` und `setProductTypeQueryDialog` erstellt werden, über die gesetzt und gelesen wird, welcher Eingabedialog für Benutzereingaben des Produkttyps verwendet werden soll.
– `QueryConfigurationDialog`
  Der Konfigurationsdialog für Benutzereingaben muss um die Konfigurationsmöglichkeiten für den neuen Datentyp ergänzt werden. Mehr dazu findet sich im Quellcode der `QueryConfigurationDialog`-Klasse.
– `QueryFactoryManager`
  Hier müssen die Konfigurationsmethoden `getProductTypeQueryInputPanelFactory` und `setProductTypeQueryInputPanelFactory` hinzugefügt werden, über die gesetzt und gelesen wird, welche Eingabepanel-Fabrik für die Erstellung von Eingabepanels für Produkttypen benutzt werden soll.
– `QueryInputPanelProducerManager`
  Über die hinzugefügten Methoden `getProductTypeQueryInputPanelProducer` und `setProductTypeQueryInputPanelProducer` wird konfiguriert, welche Eingabepanel-Producer-Klasse und damit welches Eingabepanel für die Eingabe von Produkttypen benutzt werden soll.
– `ValuesDisplayComponentProducerManager`
  Für die Einstellung, welche Werteanzeige-Komponente für die Anzeige von Werten bereits bekannter Variablen in Eingabedialogs für Produkttypen benutzt werden soll, werden die neuen Methoden `getProductTypeValuesDisplayComponentProducer` und `setProductTypeValuesDisplayComponentProducer` erstellt.

Wie wir sehen, ist eine beträchtliche Anzahl von Änderungen für die Einführung eines neuen ODL-Datentyps vorzunehmen. Nichtsdestoweniger wird der notwendige Implementierungsaufwand weitestgehend durch den Aufwand für die Erstellung des Eingabepanels für den neuen Datentyp bestimmt, weil alle anderen Änderungen mit geringem Aufwand durchführbar sind.

- **Einführung eines neuen Eingabepanels am Beispiel von `ListQueryInputPanel`:**
  Ein neues Eingabepanel zur Benutzung in Eingabedialogen kann vergleichsweise schnell in das ODL-Query-Subsystem integriert werden. Ein Eingabepanel wird als Unterklasse der abstrakten Klasse `QueryInputPanel` oder einer ihrer Unterklassen erstellt. Die zu implementierende Eingabeliste erlaubt dem Benutzer die Eingabe des Werts durch die Auswahl aus der Liste aller verfügbaren Werte – daher wird sie als Unterklasse von `SelectionQueryInputPanel` implementiert. Folgende Schritte sind zur Einführung eines Listeneingabepanels notwendig:
  - Zunächst wird die konkrete Eingabepanel-Implementierung `ListQueryInputPanel` als Unterklasse der abstrakten Klasse `SelectionQueryInputPanel` erstellt.
  - Zur Erzeugung neuer `ListQueryInputPanel`-Instanzen wird die Producer-Klasse `ListQueryInputPanelProducer` als Unterklasse von `SelectionQueryInputPanelProducer` implementiert.
  - Schließlich wird die Klasse `ListQueryInputPanel` im `QueryConfigurationDialog` in die als statisches Array ausgeführte Liste verfügbarer Eingabepanels eingetragen. Anschließend muss sie noch für jeden ODL-Datentyp, für den sie als Eingabepanel verwendet werden kann, in ein diesem Datentyp entsprechendes statisches Array eingetragen werden, das alle für diesen Datentyp einsetzbaren Eingabepanels enthält. Mehr Details zu dieser Vorgehensweise finden sich im Kommentar zum Quellcode der `QueryConfigurationDialog`-Klasse.

Abschließend wollen wir noch einige Hinweise für weitere Implementierungen geben:

- **Erweiterungen und Änderungen der ODL-Query-Subsystems:**
  Dank des modularen Aufbaus des ODL-Query-Subsystems und der Anbindung an das Auswertungssystem über eine festgelegte Schnittstelle im `QueryManager` (s. auch Abschnitt 5.2) sind Erweiterungen des Query-Subsystems ohne Änderungen an anderen ODL-Auswertungsklassen möglich, die von der Einführung neuer Eingabepanels bis hin zur Implementierung eines kompletten neuen Query-Subsystems reichen können.

  Die Query-Klassen stellen die oberste Ebene für Modifikationen am ODL-Query-Subsystem dar: muss die Benutzereingabe für einen ODL-Datentyp umfassend überarbeitet werden, so ist eine neue Query-Klasse für diesen Datentyp zu erstellen, die als *Strategie* dem `QueryManager` übergeben werden kann.

  Sind nur Änderungen am Erscheinungsbild eines Eingabedialogs oder eines Eingabepanels notwendig, so kann die entsprechende `QueryDialog`- bzw. `QueryInputPanel`-Klasse modifiziert oder eine neue erstellt werden. Damit eine neue Dialog-Klasse oder Eingabepanel-Klasse vom Benutzer zur Verwendung in Eingabedialogen ausgewählt werden kann, muss sie, wie oben beschrieben, `QueryConfigurationDialog` in die als statisches Array ausgeführte Liste verfügbarer Dialog- bzw. Eingabepanel-Klassen eingetragen werden. Nach der Durchführung dieser Schritte ist das neue Dialogfenster bzw. Eingabepanel für Benutzereingaben verfügbar und kann im Konfigurationsdialog ausgewählt werden.

- **Zukünftige Erweiterungen bei benamten Prädikaten:**
  Zurzeit können bei der Deklaration benamter Prädikate nur Sprachkonstrukte aus der CCL-Teilmenge von ODL benutzt werden, d.h, die Quantoren `context` und `new` sowie das Schlüsselwort `result` sind nicht zugelassen.

Es spricht grundsätzlich nichts dagegen, die Verwendung des vollen ODL-Sprachumfangs in benamten Prädikaten zu erlauben. Hierbei muss allerdings darauf geachtet werden, dass benamte Prädikate zurzeit in Restriktionstermen eingeschränkter Typen aufgerufen werden dürfen, in denen ebenfalls nur Sprachkonstrukte aus der CCL-Teilmenge zugelassen sind. Sollte also der Sprachumfang bei benamten Prädikaten erweitert werden, so muss kontrolliert werden, dass in Restriktionstermen eingeschränkter Typen nur solche benamten Prädikate aufgerufen werden dürfen, die lediglich die CCL-Teilmenge von ODL verwenden.

5.4 Entwurf optimierter ODL-Abfragen

In diesem Abschnitt wollen wir einige Faustregeln zum Entwurf optimierter ODL-Abfragen vorstellen. Die Optimierungsregeln berücksichtigen die technische Implementierung des ODL-Auswertungssystems und können die Auswertungszeiten für ODL-Abfragen zum Teil um eine bis mehrere Größenordnungen verkürzen. Um dem Leser die Optimierungseffekte zu veranschaulichen, werden für einige Beispielabfragen die Auswertungszeiten für eine nicht-optimierte und eine optimierte Formulierung angeben – als Testsystem wurde dabei der Rechner verwendet, auf dem diese Diplomarbeit geschrieben wurde, und als Produktmodell wurde ”FM99Fin.qml” aus dem Projekt ”FM99” verwendet, das sich im Verzeichnis ”Examples\FM99” einer QUEST-Entwicklerinstallation befindet.

- **Verwendung von Produkttypen statt Quantorlisten**
  Wird eine Universal- oder Existenzquantifizierung über mehrere Variablen benutzt, so sollten die Variablen in einem Produkttyp zusammengefasst werden. Die Abfrage

  ```
  forall ch:Channel. exists p1:Port. exists p2:Port.(
    is SourcePort( ch, p1 ) and
    is DestinationPort( ch, p2 ) )
  ```

  kann damit zu einer äquivalenten aber effizienteren Abfrage

  ```
  forall ch:Channel. exists ports:( p1:Port, p2:Port ).(
    is SourcePort( ch, ports.p1 ) and
    is DestinationPort( ch, ports.p2 ) )
  ```

  umformuliert werden. Auf dem Testsystem sank die Auswertungszeit um den Faktor 2. Ein noch drastischeres Beispiel liefert die Abfrage

  ```
  exists c1:Component. exists c2:Component.
    exists c3:Component. exists c4:Component.
    exists c5:Component.(
      c1 = c2 and c2 = c3 and c3 = c4 and c4 = c5 )
  ```

  deren optimierte Form

  ```
  exists comps:( c1:Component, c2:Component,
    c3:Component, c4:Component, c5:Component ).(
      comps.c1 = comps.c2 and comps.c2 = comps.c3 and
      comps.c3 = comps.c4 and comps.c4 = comps.c5)
  ```

  lautet. Die Auswertungszeit sinkt hierbei von 120 auf 5 Sekunden, also um den Faktor 24.

  Dieser Effekt rührt daher, dass Produkttypen zur wiederholten Iteration über die Typinstanzen der Elementtypen, die zur Erzeugung der Produkttupel notwendig ist, einen cachenden Iterator benutzen (s. auch Abschnitt 5.1.2). Der Abruf von Werten aus dem Cache ist meistens erheblich schneller als die Verwendung des Originaliterators eines Metamodelltyps, wodurch sich der Geschwindigkeitsgewinn ergibt.

  Bei dieser Optimierung muss noch darauf hingewiesen werden, dass das Einbringen der Va-

riablen, die vom ersten Quantor aus der Quantorliste gebunden wird, in den Produkttyp keine Vorteile bringt, weil über ihre Werte nur einmal iteriert werden muss – durch das in diesem Fall unnötige Cachen entsteht sogar ein geringfügiger Mehraufwand von ca. 2%.

- **Verwendung von Relationen statt Restriktionen**
  Wird eine Bedingung ausgewertet, bei der das Bestehen einer Relation zwischen Entitäten überprüft wird, so sollte die Anzahl der zu überprüfenden Entitäten nach Möglichkeit verringert werden. Manchmal lässt sich die Überprüfung der Relation ganz vermeiden, indem nicht über alle Entitäten eines Metamodelltyps zur Überprüfung der Relation iteriert wird, sondern direkt auf alle assoziierten Entitäten über den Relationsnamen zugegriffen wird. Nehmen wir als Beispiel die Abfrage

```
exists c:Component. exists pair:(ch1:Channel, ch2:Channel).(
  is Channels( c, pair.ch1 ) and is Channels( c, pair.ch2 )
  and pair.ch1.Name = ”Slot1” and pair.ch2.Name = ”Slot2” )
```

  Anstatt der Restriktion `is Channels( c, pair.ch1 )` können wir den direkten Zugriff auf alle Kanäle einer Komponente verwenden:

```
exists c:Component. exists pair:(
  ch1:element( c.Channels ), ch2:element( c.Channels ) ).(
    pair.ch1.Name = ”Slot1” and pair.ch2.Name = ”Slot2” )
```

  Die benötigte Auswertungszeit sinkt dabei um dem Faktor 30. Der Grund für den Geschwindigkeitszuwachs liegt darin, dass in der ersten Abfrage über alle Kanäle des Modells iteriert wird und erst danach die Einschränkungsbedingung angewandt wird, während die zweite Abfrage nur über die Kanäle einer Komponente iteriert, die von der mengenwertigen Assoziation `Channels` geliefert werden. Zudem müssen die Relationstests `is Channels( c, pair.ch1 )` und `is Channels( c, pair.ch2 )` nicht mehr durchgeführt werden. Der Optimierungseffekt dieser Umformulierung ist umso größer, je größer das gesamte Modell ist.

- **Teilweises Einbringen von Quantortermen als Restriktionsbedingung in den Variablentyp**
  Der von einem Existenz- oder Universalquantor gebundene Term kann teilweise als Restriktionsterm in dem Typ der vom Quantor gebundenen Variablen eingebracht werden. Das Ziel ist hierbei, die Anzahl der Typinstanzen, über die der Quantor iteriert, möglichst zu verringern und damit auch den Aufwand für die Auswertung des nicht eingebrachten Teils des Quantorterms zu senken. Betrachten wir als Beispiel die Abfrage:

```
exists p1:Port.(
  p1.Name = ”Slot1” and exists p2:Port. exists ch:Channel.(
    ch.SourcePort = p1 and ch.DestinationPort = p2 ) )
```

  Wir bringen die einfache Teilbedingung `p1.Name = ”Slot1”` aus dem Quantorterm in den Typ der Variablen `p1` ein, der dadurch zu einem eingeschränkten Typ wird:

```
exists c:Component. exists p1:{ p:Port | p.Name = ”Slot1” }.
  exists p2:Port. exists ch:Channel.(
    ch.SourcePort = p1 and ch.DestinationPort = p2 )
```

  Die Auswertungszeit sinkt durch diese Umformulierung von 50 Sekunden auf weniger als 1 Sekunde.

  Bei dieser Optimierung ist es sinnvoll, nur einen Teil des Quantorterms als Restriktionsbedingung in den Variablentyp zu verschieben. Das kann man sich daran klarmachen, dass das Einbringen des gesamten Quantorterms in den Variablentyp gar keinen Zeitgewinn bringt: der gesamte Term muss ja in diesem Fall genauso oft ausgewertet werden wie zuvor. Gleichzeitig

sollten möglichst einfache Teilbedingungen aus dem Quantorterm in den Variablentyp eingebracht werden, sodass die verbliebenen komplizierteren Bedingungen nicht für alle Typinstanzen ausgewertet werden müssen, sondern nur für diejenigen, die die einfachen Teilbedingungen erfüllen. Der Effekt dieser Optimierung hängt damit von der geschickten Aufteilung des Quantorterms in schnell auszuwertende und langsam auszuwertende Teilbedingungen ab.

- **Aufspaltung der Restriktionsbedingung bei eingeschränkten Typen**
  Bei der Aufspaltung der Restriktionsbedingungen eingeschränkter Typen handelt es sich um eine Optimierung, die auch im Abschnitt 6.2.2 besprochen wird. Solange sie nicht vom ODL-Interpreter automatisch durchgeführt werden kann, sollte sie vom Benutzer selbst vorgenommen werden.

  Wir wollen die Optimierung am folgenden Beispiel erläutern. In der Abfrage

```
exists connection:{ var:( ch:Channel, p1:Port, p2:Port ) |
  var.ch.Name = ”Slot1” and
  var.p1.Name = ”Slot1” and var.p2.Name = ”Slot” and
  var.ch.SourcePort = var.p1 and
  var.ch.DestinationPort = var.p2 }. true
```

  werden auf die Variablen `ch`, `p1` und `p2` aus dem Produkttyp `var` in der Restriktionsbedingung sowohl Restriktionen angewandt, die Abhängigkeiten zwischen den Variablen beschreiben, als auch Restriktionen, die sich nur auf eine bestimmte Variable beziehen und von anderen Variablen unabhängig sind. Letztere können in die Produkttyp-Definition verschoben werden:

```
exists connection:{ var:(
  ch:{ ch:Channel | ch.Name = ”Slot1” },
  p1:{ p1:Port | p1.Name = ”Slot1” },
  p2:{ p2:Port | p2.Name = ”Slot” }) |
    var.ch.SourcePort = var.p1 and
    var.ch.DestinationPort = var.p2 }. true
```

  Damit sinkt die Anzahl der Produkttyp-Instanzen, über die iteriert wird und auf die die verbliebene Restriktionsbedingung
  `var.ch.SourcePort = var.p1 and var.ch.DestinationPort = var.p2`
  angewandt wird. In dem obigen Beispiel verbessert sich die Auswertungszeit von 36 auf 0,1 Sekunden, was zwei Größenordnungen entspricht.

- **Verwendung einwertiger Relationen statt mehrwertiger Relationen**
  Für viele Relationen zwischen Metamodell-Entitäten gilt, dass ein und dieselbe Beziehung durch zwei symmetrische Relationen ausgedrückt wird. Zum Beispiel speichert eine Komponente in einer mehrwertigen Assoziation `SubComponents` all ihre Unterkomponenten, und gleichzeitig speichert jede Unterkomponente in der einwertigen Assoziation `SuperComponent` eine Referenz auf ihre Oberkomponente. Es gilt also:

```
forall comp:Component. forall subComp:Component.(
  is SubComponents( comp, subComp ) equiv
  subComp.SuperComponent = comp )
```

  Ausgehend von dieser Feststellung sollte man in den Fällen, wo eine Beziehung zwischen Entitäten mithilfe von verschiedenen Relationen ausgedrückt werden kann, von denen mindestens eine einwertig ist, stets die einwertige Relation verwenden. Betrachten wir die Abfrage:

```
exists var:( c1:Component, c2:Component,
  c3:Component, p:Port ).(
  neg var.c1 = var.c2 and neg var.c1 = var.c3 and
  neg var.c2 = var.c3 and (
```

```
        is Ports( var.c1, var.p ) or is Ports( var.c2, var.p ) or
        is Ports( var.c3, var.p ) ) )
```

Hier wird für den Port `p` festgestellt, ob er zu einer der paarweise verschiedenen Komponenten `c1`, `c2` oder `c3` gehört, indem die mehrwertige Relation `Ports` zwischen Komponenten und Ports benutzt wird. Die mehrwertige Relation `Ports` zwischen Komponenten und Ports kann durch die einwertige Relation `Component` zwischen Ports und Komponenten ersetzt werden:

```
exists var:( c1:Component, c2:Component,
  c3:Component, p:Port ).(
  neg var.c1 = var.c2 and neg var.c1 = var.c3 and
  neg var.c2 = var.c3 and (
    var.p.Component = var.c1 or var.p.Component = var.c2 or
    var.p.Component = var.c3 ) )
```

In diesem Beispiel sinkt die Auswertungszeit von 38 auf 32 Sekunden. Der Beschleunigungseffekt wird dadurch erreicht, dass bei der Überprüfung des Bestehens einer einwertigen Relation lediglich ein Vergleich ausgeführt werden muss, während bei der Überprüfung einer mehrwertigen Relation alle assoziierten Entitäten mit der getesteten Entität verglichen werden müssen.

Wir haben verschiedene Techniken zum Entwurf optimierter Abfragen betrachtet. Nun wollen wir ein praktisches Beispiel geben, in dem alle besprochenen Techniken Anwendung finden. Wir entwerfen ein benamtes Prädikat

```
componentsConnected( c1:Component, c2:Component )
```

der für zwei Komponenten feststellt, ob sie durch einen Kanal verbunden sind. Die Effizienz verschiedener Versionen dieses benamten Prädikats testen wir mit der Abfrage

```
exists comps:{ c:( c1:Component, c2:Component ) |
  c.c1.Name = ”Till1” and c.c2.Name = ”Connection1” }.
  call componentsConnected( comps.c1, comps.c2 )
```

die das Prädikat genau einmal ausführt. Anschließend stellen wir verschiedene Formulierungen des benamten Prädikats vor, wobei in jeder nächsten Version die Änderungen gegenüber der vorherigen durch Unterstreichung hervorgehoben werden.

Die erste unoptimierte Version des Prädikats ist

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel. exists p1:Port. exists p2:Port.(
    ( ( is SourcePort( ch, p1 ) and
        is DestinationPort( ch, p2 ) ) or
      ( is SourcePort( ch, p2 ) and
        is DestinationPort( ch, p1 ) ) ) and
    ( is Ports( c1, p1 ) and is Ports( c2, p2 ) ) )
```

Seine Ausführung dauert 68 Sekunden. Als erste Optimierung ersetzen wir die Quantorliste durch einen Produkttyp, wobei die vom ersten `exists`-Quantor gebundene Variable nicht in den Produkttyp eingebracht wird:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel. exists ports:( p1:Port, p2:Port ).(
    ( ( is SourcePort( ch, ports.p1 ) and
        is DestinationPort( ch, ports.p2 ) ) or
      ( is SourcePort( ch, ports.p2 ) and
        is DestinationPort( ch, ports.p1 ) ) ) and
    ( is Ports( c1, ports.p1 ) and is Ports( c2, ports.p2 ) ) )
```

Die Auswertungszeit sinkt dadurch auf 52 Sekunden. Als nächste Optimierung wollen wir die mehr-

wertige Relation `Ports` zwischen Komponenten und Ports durch die einwertige Relation `Component` zwischen Ports und Komponenten ersetzen:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel. exists ports:( p1:Port, p2:Port ).(
    ( ( is SourcePort( ch, ports.p1 ) and
          is DestinationPort( ch, ports.p2 ) ) or
      ( is SourcePort( ch, ports.p2 ) and
          is DestinationPort( ch, ports.p1 ) ) ) and
    ( ports.p1.Component = c1 and ports.p2.Component = c2 ) )
```

Die Auswertung dauert nun 48 Sekunden. Als nächstes bringen wir einen Teil des Quantorterms als Restriktionsbedingung in den Produkttyp ein, der dadurch zu einem eingeschränkten Typ wird:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel.( exists ports:{ p:(p1:Port, p2:Port) |
    is Ports( c1, p.p1 ) and is Ports( c2, p.p2 ) }.(
      ( is SourcePort( ch, ports.p1 ) and
          is DestinationPort( ch, ports.p2 ) ) or
      ( is SourcePort( ch, ports.p2 ) and
          is DestinationPort( ch, ports.p1 ) ) ) )
```

Die sich ergebende Auswertungszeit von 24 Sekunden lässt sich nun durch die Aufspaltung des Restriktionsterms erheblich senken:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel.( exists ports:(
    p1:{ port:Port | is Ports( c1, port ) },
    p2:{ port:Port | is Ports( c2, port ) } ).(
      ( is SourcePort( ch, ports.p1 ) and
          is DestinationPort( ch, ports.p2 ) ) or
      ( is SourcePort( ch, ports.p2 ) and
          is DestinationPort( ch, ports.p1 ) ) ) )
```

Durch diese Optimierung fällt die Auswertungszeit auf 2.7 Sekunden. Jetzt ersetzen wir die Restriktionsbedingung durch den direkten Zugriff auf die Relation `Ports` zwischen Komponenten und Ports:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ch:Channel.( exists ports:(
    p1:element (c1.Ports), p2:element (c2.Ports) ).(
      ( is SourcePort( ch, ports.p1 ) and
          is DestinationPort( ch, ports.p2 ) ) or
      ( is SourcePort( ch, ports.p2 ) and
          is DestinationPort( ch, ports.p1 ) ) ) )
```

Die Auswertungszeit verbessert sich jetzt auf 0.4 Sekunden. Als Letztes ersetzen wir die auf den Kanal bezogenen Bedingungen im Quantorterm durch den direkten Zugriff auf die Relationen `InChannel` und `OutChannels`:

```
componentsConnected( c1:Component, c2:Component ) :=
  exists ports:( p1:element (c1.Ports), p2:element (c2.Ports) ).(
    ( exists ch1:element (ports.p1.OutChannels).
        ch1 = ports.p2.InChannel ) or
    ( exists ch2:element (ports.p2.OutChannels).
        ch2 = ports.p1.InChannel ) )
```

Die nun erreichte Laufzeit beträgt gerade einmal 0.05 Sekunden, was gegenüber der Ausgangsformu-

lierung des Prädikats der Verbesserung um einen Faktor vom mehr als 1000 entspricht. Die Tabelle 5.14 fasst noch einmal die Optimierungsschritte in diesem Beispiel zusammen.

Schritt	**Optimierung**	**Laufzeit auf dem Testsystem**
–	Ausgangsformulierung	68 Sekunden
1	Produkttyp statt Quantorliste	52 Sekunden
2	Einwertige statt mehrwertige Relation	48 Sekunden
3	Teilbedingungen aus dem Quantorterm in den Variablentyp einbringen	24 Sekunden
4	Restriktionsterm aufspalten	2.7 Sekunden
5	Direkter Zugriff auf Relation statt einer Restriktion	0.4 Sekunden
6	Direkter Zugriff auf Relation statt einer Restriktion	0.05 Sekunden

Tabelle 5.14: Auswertungszeiten der Beispielabfrage nach verschiedenen Optimierungsschritten

Wie wir an dem aufgeführten Beispiel sehen, ermöglicht die konsequente Anwendung der vorgestellten Optimierungstechniken eine Laufzeitverbesserung um mehrere Größenordnungen. Zugleich muss hier darauf hingewiesen werden, dass die Zeitgewinne durch die Anwendung von Optimierungstechniken, die sich in diesem Beispiel ergeben haben, keine direkten Rückschlüsse auf die allgemeine Güte dieser Optimierungstechniken erlauben, denn die Effizienz der einzelnen Optimierungen hängt stark von der Struktur der ODL-Abfrage und von dem bearbeiteten Modell ab – ein und dieselbe Optimierung könnte in einem Fall kaum Geschwindigkeitsvorteile bringen und in einem anderen Fall die Auswertungszeit um den Faktor 10 senken. Es gilt also, in jedem konkreten Fall die anwendbaren Möglichkeiten durchzuprobieren und zu kombinieren und die beste gefundene Formulierung der ODL-Abfrage zu benutzen.

Kapitel 6

Verbesserungsmöglichkeiten

Dieses Kapitel behandelt Vorschläge für zukünftige Weiterentwicklungen des ODL-System. Sie betreffen Erweiterungen und Optimierungen des ODL-Auswertungssystems und der interaktiven Benutzerschnittstelle von ODL.

6.1 Erweiterungen des Sprachumfangs

In diesem Abschnitt befassen wir uns mit Vorschlägen für die Erweiterung der Ausdrucksmächtigkeit von ODL.

6.1.1 Teilmengen unendlicher Typen

Für Datentypen mit endlicher Domäne (z.B. `Boolean` oder `Component`) ist es möglich, Teilmengen des Typs zu verwenden, die über eine Restriktion definiert werden. So definiert die Restriktion `{c:Component | neg isEmpty( c.SubComponents )}` alle Komponenten, die mindestens eine Unterkomponente haben. Für unendliche Datentypen wie `Int` und `String` besteht zurzeit keine Möglichkeit, Teilmengen des Typs zu verwenden, da eine Restriktion der Form `{var:base_type | restriction_term}` nur dann angewendet werden kann, wenn der Basistyp iterierbar ist, was bei unendlichen Typen nicht der Fall ist.

Abhilfe kann dadurch geschaffen werden, dass endliche Teilmengen unendlicher Typen über eine feste Syntax definiert werden können. Für den Typ `Int` könnte mit `i:Int[a:b]` die Teilmenge aller ganzen Zahlen deklariert werden, die zwischen `a` und `b` liegen. Diese Syntaxerweiterung würde Abfragen wie `exists i:Int[0:10]. i < 5` ermöglichen.

6.1.2 Mengenoperationen

Für die in ODL verwendeten Mengen ist zurzeit die nur Größenbestimmung (`size(set_var)`) und die Iteration über Mengenelemente (`exists e:element set_var`) möglich. Der Sprachumfang kann für einen flexibleren Einsatz von Mengen um folgende zum Teil mengentypische Operationen erweitert werden:

- `union(set1,set2)`
  Vereinigung zweier Mengen: $M_1 \bigcup M_2 = \{ a \mid a \in M_1 \vee a \in M_2 \}$

- `intersection(set1,set2)`
  Durchschnitt zweier Mengen: $M_1 \bigcap M_2 = \{ a \mid a \in M_1 \wedge a \in M_2 \}$

- `difference(set1,set2)`
  Differenz zweier Mengen: $M_1 \backslash M_2 = \{ a \mid a \in M_1 \wedge a \notin M_2 \}$

- `asSet(value)`
  Erstellt eine Menge, die nur das Element `value` enthält: $M = \{value\}$
  Dieser Operator ist besonders dann nützlich, wenn Mengen verwendet werden müssen, die auf keinem anderen Weg deklariert werden können; beispielsweise kann eine Stringmenge `{"str1","str2"}` über den Ausdruck

  ```
  union( asSet("str1"), asSet("str2") )
  ```

  erhalten werden. Eine näherliegende Deklaration

  ```
  { s:String | s = "str1" or s = "str2" }
  ```

  ist nicht zulässig, da über die Werte des unendlichen Typs `String` nicht iteriert werden kann (und auch, wenn dies möglich wäre, so wäre diese Deklaration u.U. sehr ineffizient).

- `unite( var:type, expression( var ) )`
  Punktweise Auswertung eines Ausdrucks: für alle Belegungen von `var` wird der Ausdruck `expression` ausgewertet und alle Ergebnisse zu einer Menge vereinigt:

  $$\texttt{unite(var:type,expression(var))} = \bigcup_{\texttt{var} \in \texttt{type}} \texttt{expression(var)}$$

  Wie der Operator `asSet` ist dieser Operator nützlich, um Mengen zu deklarieren, die nicht über die Restriktion eines Datentyps erhalten werden können. Beispielsweise kann man dem Benutzer alle Komponentennamen zur Auswahl anbieten:

  ```
  context compName: unite( c:Component, asSet( c.Name ) ).true
  ```

  Die Abfrage

  ```
  context compName:{ s:String |
    exists c:Component. c.Name = s}. true
  ```

  über die Restriktion des Typs `String` sieht auf den ersten Blick äquivalent aus, hat aber eine ganz andere Wirkung: anstatt die Namen aller Komponenten in einer Liste zur Auswahl anzubieten, wird im Eingabedialog für diese Abfrage lediglich ein Eingabefeld für den String `s` angezeigt – die Eingabe kann hier nur dann abgeschlossen werden, wenn der eingegebene String gleich dem Namen einer existierenden Komponente ist.

Die Operatoren `union`, `intersection` und `difference` sind mit geringem bis mittlerem Aufwand implementierbar. Der Operator `asSet` ist sehr einfach zu implementieren. Die Implementierung von `unite` ist von mittlerer Schwierigkeit.

6.1.3 Dynamische Informationen in `context`-Abfragedialogen

In `context`-Abfragen kann in der aktuellen Version ein fest vorgegebener Hinweistext angezeigt werden. Eine Verbesserungsmöglichkeit würde die Anzeige dynamischer Informationen darstellen, die von den aktuellen Variablenbelegungen abhängen, beispielsweise würde im Eingabedialog für die Abfrage

```
context [ Hint="Hinweistext" Info="Anzahl der ausgewählten
  Ports:": size( portSet )] portSet:set Port. true
```

neben dem statischen Hinweistext auch die Größe der Portmenge angezeigt, die sich während der Eingabe dynamisch ändern kann.

6.1.4 Arithmetische Division

Bei der Auswertung von ODL-Abfragen ist es von essentieller Bedeutung, dass alle ODL-Terme ein definiertes Ergebnis zurückgeben. Insbesondere dürfen auch die arithmetischen Ausdrücke nur totale Operationen verwenden. Genau aus diesem Grund unterstützt ODL zurzeit nur die arithmetischen Operationen Addition, Subtraktion und Multiplikation. Da die Division auf reellen Zahlen nur partiell definiert ist, wurde sie bis jetzt nicht in den ODL-Sprachumfang aufgenommen.

Damit die Division in ODL verwendet werden kann, muss sie totalisiert werden. Hierfür muss der Fall der Division durch Null gesondert behandelt werden: ein ODL-Ausdrück soll auch dann ausgewertet werden können, wenn im Laufe der Auswertung eine Division durch Null auftritt.

Um die Division zu totalisieren, erweitern wir ihre Wertemenge um den Wert `undef`, der immer dann als Ergebnis eines Ausdrucks zurückgegeben wird, wenn das Ergebnis nicht definiert ist. Es gilt dann $\texttt{div:Int} \times \texttt{Int} \rightarrow \texttt{Int} \bigcup \{\texttt{undef}\}$, wobei $\texttt{a}/0 = \texttt{undef}$ für alle $\texttt{a} \in \texttt{Int}$ ist.

Ergibt ein arithmetischer Ausdruck anstatt einer Zahl den Wert `undef`, so muss dieser bei der Auswertung des ODL-Terms besonders behandelt werden. Da arithmetische Ausdrücke in ODL nur innerhalb von booleschen Termen vorkommen dürfen (s. Abschnitt 4.1), muss bei booleschen Termen die Möglichkeit berücksichtigt werden, dass das Ergebnis eines Operanden `undef` ist. Dabei entsteht folgendes Problem: das Ergebnis eines booleschen Terms kann darüber entscheiden, ob eine bestimmte Transformation am Modell durchgeführt wird oder nicht. Hierbei kann sowohl das Ergebnis `true` als auch das Ergebnis `false` zu einer Transformation führen, abhängig davon, welches Termergebnis das gewünschte ist. Wir wollen das am Beispiel zweier Abfragen demonstrieren:

- ```
  exists c:Component.(
    size( c.Channels ) = 0 and
    result has Name( c, ”NoChannels” ) )
  ```
  Diese Abfrage benennt alle Komponenten um, die keine Kanäle besitzen: eine Modelltransformation wird durchgeführt, wenn der Term `size( c.Channels ) = 0` zu `true` ausgewertet wird. Das gewünschte Ergebnis, das zur Modelltransformation führt, ist hier also `true`.

- ```
  exists c:Component.(
    neg size( c.Channels ) > 0 and
    result has Name( c, ”NoChannels” ) )
  ```
  Diese Abfrage has diegleiche Auswirkung, wie die vorherige – nur wird hier eine Modelltransformation dann durchgeführt, wenn der Term `size( c.Channels ) > 0` zu `false` evaluiert. Damit ist das gewünschte Ergebnis in diesem Fall `false`.

Ein undefiniertes Ergebnis eines booleschen Terms darf nie zu einer Modelltransformation führen, denn ein undefiniertes Ergebnis ist in Wirklichkeit eine Exception im Laufe der Auswertung. Wie man an dem angeführten Beispiel sieht, kann das Ergebnis `undef` weder wie `true` noch wie `false` behandelt werden, denn beide Ergebnisse können eine Änderung am Modell nach sich ziehen. Damit ist zur korrekten Behandlung undefinierter Ergebnisse von booleschen Termen die Erweiterung der aktuell verwendeten Logik notwendig.

Eine Möglichkeit besteht in der Verwendung einer *dreiwertigen Logik*, wie sie beispielsweise in SQL eingesetzt wird ([Kemper], S.110-111). Sie enthält neben den üblichen booleschen Werten `true` und `false` den Wert `undef` (diesmal als logischen Wert) – dieser Wert tritt als Ergebnis eines booleschen Ausdrucks auf, dessen Auswertung nicht erfolgreich durchgeführt werden konnte und dessen Ergebnis somit undefiniert ist. Für die dreiwertige Logik gelten die in Tabelle 6.1 aufgeführten Wahrheitstabellen. Die Verknüpfungen Implikation und Äquivalenz lassen sich auf die Verpknüpfungen `AND`, `OR` und `NOT` zurückführen, ihre Wahrheitstabellen werden aber der Vollständigkeit halber angegeben.

Der Auswertungsalgorithmus für ODL-Ausdrücke bleibt für die Fälle, wo ein boolescher Term zu `true` oder `false` ausgewertet wird, unverändert. Liefert ein Term als Ergebnis `undef` zurück, so hängt die Vorgehensweise von dem aufrufenden Term ab:

NOT	**true**	**false**	**undef**
	false	true	undef

AND	**true**	**false**	**undef**
true	true	false	undef
false	false	false	false
undef	undef	false	undef

OR	**true**	**false**	**undef**
true	true	true	true
false	true	false	undef
undef	true	undef	undef

$\Longrightarrow$	**true**	**false**	**undef**
true	true	false	undef
false	true	true	true
undef	true	undef	undef

$\Longleftrightarrow$	**true**	**false**	**undef**
true	true	false	undef
false	false	true	undef
undef	undef	undef	undef

Tabelle 6.1: Wahrheitstabellen für dreiwertige Logik

- **Logischer Operator**
  Ist der aufrufende Term ein logischer Operator (z.B. `(term1 and term2)` oder `(neg term1)`), so wird zur Auswertung die dem Operator entsprechende Wahrheitstabelle aus der Tabelle 6.1 herangezogen.

- **Gleichheit**
  Wird in einer Gleichheit `(expr1 = expr2)` genau einer der zu vergleichenden Ausdrücke zu `undef` ausgewertet, so ist das Ergebnis der Gleichheit `false`, weil das Ergebnis des jeweils anderen Ausdrucks ungleich `undef` und damit ungleich dem Ergebnis des ersten Ausdrucks ist.

  Werden in einer Gleichheit beide Ausdrücke zu `undef` ausgewertet, so ist auch das Ergebnis der Gleichheit `undef`, weil für zwei Ausdrücke mit undefinierten Ergebnissen nicht ermittelt werden kann, ob ihre Ergebnisse gleich sind.

- **Quantor**
  Bei Quantoren muss man folgende Fälle unterscheiden:

  – Quantoren `context` und `new`
    Für diese Quantoren, d.h., in einem Ausdruck der Form `context var:type.term` oder `new var:type.term`, wird einfach das Ergebnis des Quantorterms weitergegeben – wenn der Term zu `undef` ausgewertet wird, so ist auch das Ergebnis des Quantors gleich `undef`.
    Hierbei ist es unproblematisch, wenn der von einem `new`-Quantor gebundene Term zu `undef` evaluiert, während ein neues Modellelement vom `new`-Quantor bereits erstellt wurde: die Änderung am Modell findet erst statt, wenn das neue Modellelement in das Modell eingefügt wird, und dies wird nicht passieren, falls der vom `new`-Quantor gebundener Term zu `undef` evaluierte.

  – Quantor `forall`
    Nach der Definition des Universalquantors gilt

$$\texttt{forall var:type.term(var)} = \bigwedge_{\texttt{var} \in \texttt{type}} \texttt{term(var)}$$

    Ausgehend von dieser Definition und von den Wahrheitstabellen in der Tabelle 6.1 ist das Ergebnis eines Universalquantors gleich:

  * `true`, wenn gilt
    $\forall$ `var` $\in$ `type: term(var)=true`
  * `false`, wenn gilt
    $\exists$ `var` $\in$ `type: term(var)=false`
  * `undef`, wenn gilt
    $\exists$ `var` $\in$ `type: term(var)=undef` $\wedge$
    $\nexists$ `var` $\in$ `type: term(var)=false`

- Quantor `exists`
  Nach der Definition des Existenzquantors gilt

$$\texttt{exists var:type.term(var)} = \bigvee_{\texttt{var}\in\texttt{type}} \texttt{term(var)}$$

  Ausgehend von dieser Definition und von den Wahrheitstabellen in der Tabelle 6.1 ist das Ergebnis eines Existenzquantors gleich:

  * `true`, wenn gilt
    $\exists$ `var` $\in$ `type: term(var)=true`
  * `false`, wenn gilt
    $\forall$ `var` $\in$ `type: term(var)=false`
  * `undef`, wenn gilt
    $\exists$ `var` $\in$ `type: term(var)=undef` $\wedge$
    $\nexists$ `var` $\in$ `type: term(var)=true`

Bei der Auswertung von Quantortermen ist noch die Besonderheit zu berücksichtigen, dass die Quantoren als Ergebnis neben dem logischen Wert auch die Menge aller erfüllenden Belegungen für die quantifizierte Variable zurückgeben – so könnte die Abfrage

```
exists c:Component. true
```

beispielsweise folgendes Ergebnis liefern:

{ true, (c, firstComp), (c, secondComp), (c, thirdComp) }.

Dementsprechend müssen wir die Regeln definieren, nach denen die erfüllenden Belegungen ermittelt werden, falls der vom Quantor gebundene Term zu `undef` ausgewertet wurde. Dies ist aber kein besonderes Problem, denn die Menge der erfüllenden Belegungen ist nichts anderen als die Mengen aller Variablenbelegungen, für die der vom Quantor gebundene Term das gewünschte Ergebnis aus der Menge {`true`, `false`} liefert. Demnach werden alle Belegungen, für die der Term zu `undef` evaluiert, von den erfüllenden Belegungen ausgeschlossen. Der Auswertungsalgorithmus muss damit in Bezug auf die Ermittlung der Menge der erfüllenden Belegungen nicht modifiziert werden, weil er, wie auch in der aktuellen Implementierung, nur die Belegungen berücksichtigen soll, für die das Ergebnis der Termauswertung `true` oder `false` ist.

6.2 Optimierung der Abfrageauswertung

In diesem Abschnitt besprechen wir mögliche Optimierungen des ODL-Auswertungssystems.

6.2.1 Erweiterung der Skolem-Optimierung für `context`- und `new`-Quantoren

In der aktuellen Version werden `context`- und `new`-Quantoren in einer ODL-Abfrage bis zum nächsten `forall`-Quantor oder – falls keine `forall`-Quantoren vor dem betreffenden `context`- bzw. `new`-Quantor vorkommen – an den Anfang der Abfrage vorgezogen (s. auch [Pasch], S.38-39). Deshalb werden wir der Kürze halber `context`- und `new`-Quantoren im Weiteren unter dem

Begriff *verschiebbare Quantoren* zusammenfassen. Außerdem bezeichnen wir für einen verschiebbaren Quantor den vorausgehenden `forall`-Quantor oder – falls keiner vorhanden – den Anfang der ODL-Abfrage als *Verschiebungsgrenze*. Wir wollen die Verschiebungssregel an einem Beispiel veranschaulichen. Die Abfrage

```
exists p1:Port. context p2:Port. forall p3:Port.
  exists p4:Port. new p5:Port. context p6:Port. true
```

wird vom ODL-Interpreter wie folgt umgeformt:

```
context p2:Port. exists p1:Port. forall p3:Port.
  new p5:Port. context p6:Port. exists p4:Port. true
```

Diese Regel gilt, solange die vom verschiebbaren Quantor gebundene Variable nicht von anderen Variablen abhängt. Solche Abhängigkeiten können bei `RestrictedType`-Variablen und bei Mengenvariablen eintreten. Diese Einschränkung betrifft allerdings nur `context`-Quantoren, weil `new`-Quantoren nur Variablen eines Metamodelltyps (z.B. `Component`, `Port`, etc.), und damit nur unabhängige Variablen binden können.

Nehmen wir als Beispiel für diese Einschränkung bei `context`-Quantoren zwei vom Ergebnis her äquivalente Abfragen, in denen für jede existierende Komponente eine Unterkomponente auszuwählen ist:

- Auswahl aus der Menge der Unterkomponenten:

```
exists comp:Component.
  context subComp:element( comp.SubComponents ). true
```

- Auswahl über einen eingeschränkten Typ:

```
exists comp:Component.
  context subComp:{ c:Component |
    is SubComponents( comp, c ) }. true
```

In beiden Fällen ist die Variable `subComp` abhängig von der Variablen `comp`, sodass der zu `subComp` gehörende `context`-Quantor nicht an den Anfang der Abfrage vorgezogen werden kann.

In der jetzigen Version des ODL-Interpreters wird ein `context`-Quantor, dessen Variable von anderen Variablen abhängig ist, als unbeweglich markiert und bei der Skolem-Optimierung an seiner Stelle gelassen. Dies ist auch dann der Fall, wenn der Quantor theoretisch vorgezogen werden könnte. In der Abfrage

```
context comp:Component. exists port:Port.
  context subComp:element(comp.SubComponents). true          (6.1)
```

kann der Quantor `context subComp` durchaus vor den Quantor `exists port` vorgezogen werden, weil die Variable `subComp` nicht von der Belegung der Variablen `port` abhängt. Dies wird zurzeit der Einfachheit halber unterlassen.

Die Nachbedingungen des aktuellen Algorithmus sind die folgenden:

- Ist die von einem verschiebbaren Quantor *q1* gebundene Variable von keiner anderen Variablen abhängig, so befindet sich dieser Quantor unmittlerbar hinter seiner Verschiebungsgrenze. Dabei dürfen sich zwischen *q1* und seiner Verschiebungsgrenze weitere verschiebbare Quantoren befinden, falls sie sich bereits in der ursprünglichen ODL-Abfrage vor *q1* befanden.

- Ist die von einem `context`-Quantor *q1* gebundene Variable von anderen Variablen abhängig, so befindet sich der `context`-Quantor an derselben Stelle wie in der ursprünglichen ODL-Abfrage.

- Die Reihenfolge von verschiebbaren Quantoren, die im Laufe des Algorithmus verschoben wurden, bleibt in Bezug aufeinander beibehalten.

Im Zuge der weiteren Entwicklung des ODL-Interpreters kann die Positionierung von verschiebbaren Quantoren innerhalb einer ODL-Abfrage so verfeinert werden, dass folgende Nachbedingungen für jeden verschiebbaren Quantor nach der Umformung der ODL-Abfrage gelten:

- Die erste Nachbedingung stimmt mit der oben angegebenen ersten Nachbedingung des aktuellen Algorithmus überein: ein verschiebbarer Quantor, dessen Variable von keiner anderen Variablen abhängt, befindet sich direkt hinter seiner Verschiebungsgrenze, wobei sich weitere verschiebbare Quantoren zwischen ihm und seiner Verschiebungsgrenze befinden dürfen.

- Ist die von einem `context`-Quantor *q1* gebundene Variable von anderen Variablen abhängig, so befindet sich *q1* an der frühestmöglichen Position hinter seiner Verschiebungsgrenze und allen Quantoren, die Variablen binden, von denen die von *q1* gebundene Variable abhängt.

- Befand sich ein `context`-Quantor *q1* im ursprünglichen ODL-Ausdruck hinter einem anderen `context`-Quantor *q2*, so darf sich *q1* in der umgeformten ODL-Abfrage nur dann vor *q2* befinden, falls *q2* hinter einem Quantor *q* steht, von dessen Variablen die Variable von *q2* abhängig und die Variable von *q1* unabhängig ist (*q* ist dabei kein `forall`-Quantor).

  Diese Bedingung gewährleistet, dass die ursprüngliche Reihenfolge der verschiebbaren Quantoren untereinander soweit wie möglich beibehalten wird.

Der wesentliche Unterschied zu dem jetzigen Algorithmus besteht also darin, dass ein `context`-Quantor *q1*, dessen Variable von anderen Variablen abhängt, nicht mehr zwingend an seiner Stellen bleiben muss, sondern bis zu einer früheren Position vorgezogen werden kann, die sich hinter allen Quantoren befindet, von deren Variablen die Variable von *q1* abhängt.

Der Algorithmus 1 nimmt die Umformung einer ODL-Abfrage bezüglich der Positionen verschiebbarer Quantoren vor, sodass die oben aufgezählten Nachbedingungen nach der Ausführung des Algorithmus gelten.

> Algorithmusidee:
> Verschiebbare Quantoren können nur zwischen zwei Verschiebungsgrenzen bewegt werden, weil kein verschiebbarer Quantor über eine Verschiebungsgrenze hinweg bewegt werden kann. Die Hauptschleife des Algorithmus ruft deshalb für jede Verschiebungsgrenze in der Abfrage (außer der letzten, die entweder das Ende der Abfrage oder ein am Ende der Abfrage stehender `forall`-Quantor ist) eine innere Schleife auf, die verschiebbare Quantoren innerhalb des Intervalls zwischen der betrachteten und der nachfolgenden Verschiebungsgrenze neu positioniert.
> Die innere Schleife betrachtet jeden verschiebbaren Quantor zwischen der aktuellen und der nächsten Verschiebungsgrenze: zunächst wird der Quantor nach vorne verschoben, bis er auf die Verschiebungsgrenze oder auf einen Quantor trifft, von dessen Variablen er abhängig ist. Befindet sich nach dieser Stelle eine Gruppe verschiebbarer Quantoren, so wird der betrachtete verschiebbare Quantor zurück hinter diese Gruppe verschoben, jedoch nicht weiter als an seine ursprüngliche Position. Damit wird gewährleistet, dass die Reihenfolge verschiebbarer Quantoren in Bezug aufeinander soweit wie möglich erhalten bleibt.

Solange dieser Optimierungsvorschlag nicht implementiert ist, kann seine Wirkung vom Benutzer durch gezielte Positionierung von `context`-Quantoren nachgeahmt werden. So würde in der aktuellen Version des ODL-Übersetzers die Abfrage 6.1 nicht umgeformt und der Quantor `context subComp` nicht vor den Quantor `exists port` vorgezogen. Dies führt dazu, dass für jeden Port der `context`-Eingabedialog für die Variable `subComp` gestartet wird, obwohl dies nur für jede Belegung von `comp`, nicht aber für jede Belegung von `port` nötig ist. Der Benutzer kann hier die Abfrage explizit so umformulieren, wie dies auch von dem vorgestellten Positionierungsalgorithmus geleistet würde:

```
context comp:Component.
  context subComp:element( comp.SubComponents ).        (6.2)
    exists port:Port. true
```

Algorithmus 1 Neupositionierung verschiebbarer Quantoren

```
quantorList := Liste aller Quantoren aus der ODL-Abfrage in der Reihenfolge ihres Auftretens
if not quantorList.isEmpty then
  firstQuantor := quantorList.first()
  co Markierungen für den Anfang und das Ende der ODL-Abfrage in quantorList einfügen oc
  quantorList.insertAsFirst( START_MARK )
  quantorList.insertAsLast( FINISH_MARK )
else
  firstQuantor := null
end if
leftLimit := START_MARK
co Hauptschleife oc
while firstQuantor ≠ null do
  if Ein forall-Quantor ist hinter firstQuantor vorhanden then
    rightLimit := Gefundener nächster forall-Quantor hinter firstQuantor
  else
    rightLimit := FINISH_MARK
  end if
  co Quantoren zwischen leftLimit und rightLimit neu positionieren oc
  quantor := quantorList.nextAfter( leftLimit )
  while quantor ≠ rightLimit do
    co Position für quantor finden oc
    posQuantor := quantorList.previousBefore( quantor )
    co  Bis zur linken Verschiebungsgrenze oder bis zu einem Quantor zurückgehen, von dessen Variablen
    quantor.variable abhängig ist oc
    while posQuantor ≠ leftLimit and (quantor.variable ist von posQuantor.variable unabhängig) do
      posQuantor := quantorList.previousBefore( posQuantor )
    end while
    nextPosQuantor := quantorList.nextAfter(posQuantor)
    co Wenn nach posQuantor eine Gruppe verschiebbarer Quantoren folgt, muss quantor ans Ende dieser
    Gruppe verschoben werden, damit die Reihenfolge verschiebbarer Quantoren untereinander erhalten
    bleibt, jedoch nicht weiter als an seine ursprüngliche Position oc
    while (nextPosQuantor ist ein verschiebbarer Quantor) and (nextPosQuantor ≠ quantor) do
      nextPosQuantor := quantorList.nextAfter( nextPosQuantor )
    end while
    co nextPosQuantor gibt nun den Quantor an, vor den quantor verschoben werden muss oc
    if nextPosQuantor ≠ quantor then
      quantorList.remove( quantor )
      quantorList.insertBefore( nextPosQuantor, quantor ) co quantor vor nextPosQuantor in die Liste ein-
      fügen oc
    end if
    quantor := quantorList.nextAfter( quantor )
  end while
  if (Nicht-forall-Quantoren hinter rightLimit in quantorList vorhanden) then
    firstQuantor := Erster Nicht-forall-Quantor hinter rightLimit
    leftLimit := quantorList.previousBefore( firstQuantor )
  else
    firstQuantor := null
  end if
end while
co listQuantor enthält nun die optimierte Reihenfolge der Quantoren in der ODL-Abfrage oc
```

Durch geeignete Formulierung von ODL-Abfragen kann also erreicht werden, dass der beschriebene verfeinerte Algorithmus keine weiteren Optimierungen gegenüber dem aktuell implementierten Algorithmus erreicht. Nichtsdestoweniger ist die Implementierung dieses Algorithmus von Vorteil, denn er nimmt dem Benutzer die Notwendigkeit ab, auf die optimale Positionierung verschiebbarer Quantoren in ODL-Abfragen zu achten, indem er jede Abfrage automatisch zu einer äquivalenten Abfrage mit optimal positionierten verschiebbaren Quantoren umwandelt.

6.2.2 Optimierung eingeschränkter Typen

Wir wollen uns in diesem Abschnitt mit der Optimierung der Iteration über Instanzen eingeschränkter Typen beschäftigen, deren Basistypen aus mindestens zwei Elementtypen besteht. Bei einem solchen eingeschränkten Typ kann der Restriktionsterm Bedingungen enthalten, für deren Berechnung nur eine Teilmenge aller Typelemente des Basistyps verwendet wird. Betrachtet wir einen als Beispiel einen eingeschränkten Typ der Form

```
{p:( x1:Type1, x2:Type2 ) | P(p.x1) and Q(p.x1, p.x2)}        (6.3)
```

wobei `P` und `Q` ODL-Terme sind. Hier muss die Restriktionsbedingung `P` nicht für alle Belegungen von `x2` neu berechnet werden, da `x2` keinen Einfluss auf ihr Ergebnis hat. Bei einer nicht-optimierten Iteration über die Werte dieses eingeschränkten Typs würde die Bedingung jedoch für jedes Tupel `(x1,x2)` $\in$ `Type1`$\times$`Type2` und damit für jede Belegung von `x2` ausgewertet.

Eine Optimierung bestünde darin, dass der Basistyp `(x1:Type1,x2:Type2)` in seine Elemente `x1:Type1` und `x2:Type2` aufgeteilt wird und die Restriktionsbedingung nur auf das Element `x1` angewandt wird. Dies kann mit folgender Typsubstitution erreicht werden:

```
{p:( x1:{x1:Type1|P(x1)}, x2:Type2 ) | Q(p.x1, p.x2)}        (6.4)
```

Wir wollen den Zeitgewinn durch die Optimierung berechnen. Nehmen wir an, dass die Restriktionsbedingung `P` die Zeit t_p und die Restriktionsbedingung `Q` die Zeit t_q zur Auswertung benötigt. Ferner sei $M_1 = |\texttt{Type1}|$ und $M_2 = |\texttt{Type2}|$. Außerdem seien t_1 und t_2 jeweils die Zeiten, die für das Holen einer Instanz des Typs `Type1` bzw. `Type2` benötigt werden. Schließlich sei M_1^P die Anzahl der Instanzen von `Type1`, welche die Bedingung `P` erfüllen: es gilt natürlich $M_1^P \leq M_1$ und oft sogar $M_1^P << M_1$. Dann brauchen wir folgende Zeiten für die Iteration über alle Instanzen des obigen eingeschränkten Typs:

- Nicht-optimierte Formulierung:
  $T_1 = t_1 * M_1 + t_p * M_1 * M_2 + (t_2 + t_q) * M_1 * M_2$

- Optimierte Formulierung:
  $T_2 = t_1 * M_1 + t_p * M_1 + (t_2 + t_q) * M_1^P * M_2$

Der Zeitgewinn bei der optimierten Version gegenüber der nicht-optimierten Version beträgt hier
$T_2 - T_1 = t_p * M_1 * (M_2 - 1) + (t_2 + t_q) * \left(M_1 - M_1^P\right) * M_2$
Die Einsparung kommt dadurch zustande, dass die Bedingung `P` nur M_1-mal statt $M_1 * M_2$-mal ausgewertet wird, und das Holen eines Werts von `Type2` und die anschließende Berechnung von `Q` nicht mehr $M_1 * M_2$-mal, sondern $M_1^P * M_2$ stattfindet. Der Zeitgewinn ist also umso größer, je weniger Instanzen von `Type1` die Bedingung `P` erfüllen, d.h., je stärker `P` die Instanzen von `Type1` filtert.

Die beschriebene Optimierungstechnik findet breite Anwendung in Datenbanken, um den Aufwand für Selektionen zu senken, die auf Kreuzprodukten mehrerer Tabellen stattfinden. Im Abschnitt 3.1 haben wir bereits die Ähnlichkeit zwischen ODL und SQL angesprochen. Tatsächlich kann man eine ODL-Abfrage der Form

```
exists tuple:{ p:( x1:Type1, x2:Type2 ) |
  P(p.x1) and Q(p.x1,p.x2) }. true
```
(6.5)

leicht in eine SQL-Abfrage übertragen, wobei Tabellen die Rolle der ODL-Datentypen spielen, Records die Rolle der Typinstanzen erfüllen, und die Attribute `x1` bzw. `x2` in den Records der Tabellen `Type1` bzw. `Type2` die Variablenwerte für `x1` bzw. `x2` repräsentieren:

```
select x1, x2
from Type1, Type2
where P and Q
```

Aus der datenbanktechnischen Sicht handelt es sich bei dieser Abfrage um die Bildung eines Kreuzproduktes zweier Tabellen mit anschließender Selektion mit der Bedingung `P and Q`. Der algebraische Ausdruck für diese SQL-Abfrage lautet demnach

$$\sigma_{\mathtt{P} \wedge \mathtt{Q}} (\mathtt{Type1} \times \mathtt{Type2})$$

Da `P` nur Attribute der Tabelle `Type1` (nämlich das Attribut `x1`) enthält, lässt sich dieser Ausdruck nach den Regeln der relationalen Algebra zum folgenden Ausdruck umformen:

$$\sigma_{\mathtt{Q}} (\sigma_{\mathtt{P}} (\mathtt{Type1}) \times \mathtt{Type2})$$

Eine entsprechende SQL-Abfrage würde wie folgt aussehen:

```
select x1, x2
from ( select x1 from Type1 where P ), Type2
where Q
```

Wenn wir dieser SQL-Abfrage in eine analoge ODL-Abfrage übersetzen, so erhalten wir genau die optimierte Formulierung der Abfrage 6.5:

```
exists tuple:{ p:( x1:{x1:Type1|P(x1)}, x2:Type2 ) |
  Q(p.x1, p.x2) }. true
```
(6.6)

Wir sehen also, dass sich Techniken zur logischen Optimierung von SQL-Abfragen auf die Optimierung von ODL-Abfragen übertragen lassen. Weitere Informationen zur logischen Optimierung bei der Auswertung von SQL-Abfragen finden sich in [Kemper] (S.206-214).

Betrachten wir nun ein konkretes Beispiel:

```
exists var:{ v:( p1:Port, p2:Port, ch:Channel ) |
 ( v.p1.Name = ”Slot1” or
   v.p1.Name = ”Slot2” /* Cond1 */ ) and
 v.p2.Name = ”Slot” /* Cond2 */ and
 v.ch.SourcePort = v.p1 and
 v.ch.DestinationPort = v.p2 /* Cond3 */ }. true
```
(6.7)

Im Restriktionsterm sind alle Bedingungen mit dem logischen `AND` verknüpft, sodass wir den Restriktionsterm aufbrechen und Teilbedingungen auf einzelne Elemente des Basistyps anwenden dürfen. Durch diese Umformulierung ergibt sich eine äquivalente aber viel schnellere ODL-Abfrage:

```
exists var:{ v:(
 p1:{ p1:Port | p1.Name = ”Slot1” or
   p1.Name = ”Slot2” /* Cond1 */ },
 p2:{ p2:Port | p2.Name = ”Slot” /* Cond2 */ },
 ch:Channel ) |
   v.ch.SourcePort = v.p1 and
   v.ch.DestinationPort = v.p2 /* Cond3 */ }. true
```
(6.8)

Die Bedingungen `Cond1` und `Cond2` werden hier nur auf die Typelemente angewandt, die zur Auswertung benötigt werden; nur die Bedingung `Cond3`, die alle Typelemente referenziert, verblieb im Restriktionsterm des äußeren eingeschränkten Typs.

Nun wollen wir die allgemeine Idee des Algorithmus zur Durchführung der beschriebenen Optimierung vorstellen (eine ausführliche Beschreibung würde über der Rahmen dieser Diplomarbeit hinausgehen). Gegeben sei ein eingeschränkter Typ der Form

```
{var:(x1:T1,x2:T2,...,xn:Tn) | RestrictionTerm(var.x1,...,var.xn)}
```

Der Kürze halber werden wir bei der Algorithmusbeschreibung von der ODL-Notation geringfügig abweichen und einfach $\{var : (x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) \mid P_{main}(x_1, x_2\ \ldots, x_n)\}$ schreiben (hier wird auf die in ODL-Abfragen notwendige Verwendung von Selektoren verzichtet). Der Algorithmus besteht aus folgenden Schritten:

1: Umwandlung von Äquivalenzen und Implikationen in Terme, die nur das logische `AND`, `OR` und `NEG` verwenden.

2: Anwendung von DeMorgans Gesetz auf Disjunktionen: alle Terme der Form $\neg(p_1 \vee p_2)$ werden zu $\neg p_1 \wedge \neg p_2$ umgewandelt.

3: Überprüfung, ob ungeklammerte `OR`-Verknüpfungen im Restriktionsterm vorkommen, d.h., ob der Term die Form $p_1 \wedge p_2 \wedge \ldots \wedge p_i \vee p_{i+1} \wedge \ldots$ hat. Wenn ja, so ist eine Optimierung nicht möglich und der Algorithmus bricht ab.

   Hier ist darauf zu achten, dass nur ungeklammerte `OR`-Verknüpfungen kritisch sind; der Optimierung eines Term der Form $p_1 \wedge (p_2 \vee p_3)$ steht nichts im Wege, da die `OR`-Verknüpfung sich innerhalb eines Teilterms und nicht im Hauptterm befindet.

   Falls der Algorithmus nach diesem Schritt fortgesetzt wird, so ist sichergestellt, dass der Restriktionsterm die Form $p_1(L_1) \wedge p_2(L_2) \wedge \ldots \wedge p_m(L_m)$ hat, wobei L_i Listen der Basistyp-Elemente sind, die von den Prädikaten p_i als Argumente verwendet werden, d.h. $L_i = \mathrm{x}_{k_{i,1}}, \mathrm{x}_{k_{i,2}}, \ldots, \mathrm{x}_{k_{i,r_i}}$ mit $k_{i,j} \in \{1, \ldots, n\}$.

4: Aufspaltung des Restriktionsterms in Teilterme, die durch das logische `AND` verknüpft sind. Diese Terme werden in Listen $M_1, M_2, \ldots, M_n$ gespeichert, wobei die Liste M_i alle Terme enthält, die genau i Typelemente aus dem Basistyp verwendet: $\mathrm{P}_i \in M_t \ \Leftrightarrow\ |\mathrm{L}_i| = t$.

5: Verpacken von Teiltermen mit den von ihnen verwendeten Basistyp-Elementen in separaten eingeschränkten Typen, die als Elementtypen des Basistyps dienen werden. Dieser Schritt bildet den Hauptteil des Algorithmus:

   a) Bringe alle Prädikate, die nur ein Argument haben, d.h., alle $p_i \in L_1$, mit ihrem Argument in einem eingeschränkten Elementtyp zusammen. Sind beispielsweise $p_1(x_1)$ und $p_2(x_3)$ solche Prädikate, so wird der Typ
      $\{x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n \mid p_1(x_1) \wedge p_2(x_3) \wedge p_3(L_3) \wedge \ldots \wedge p_m(L_m)\}$
      zu
      $\{var_1 : \{x_1 : T_1|p_1(x_1)\}, x_2 : T_2, var_3 : \{x_3 : T_3|p_2(x_3)\}, \ldots, x_n : T_n) \mid$
      $p_3(L_3) \wedge p_4(L_4) \wedge \ldots \wedge p_m(L_m)\}$
      umgewandelt.

   b) Bringe Prädikate mit mehreren Argumenten, soweit möglich, mit diesen Argumenten in einem eingeschränkten Elementtyp zusammen.

      Dieser Schritt ist schwieriger als Schritt 5a, weil es in vielen Situationen keine eindeutige Aufteilung des Basistyps gibt. Ein einfaches Beispiel dafür ist der Typ
      $\{var : (x_1 : T_1, x_2 : T_2, x_3 : T_3) \mid p_1(x_1, x_2) \wedge p_2(x_2, x_3)\}$:

gruppiert man p_1 mit dem Typ $(x_1 : T_1, x_2 : T_2)$, so lässt sich der entstehende Typ
$\{var : (v_{12} : \{v_{12} : (x_1 : T_1, x_2 : T_2) \mid p_1(x_1, x_2)\}, x_3 : T_3) \mid p_2(x_2, x_3)\}$
nicht weiter optimieren. Analog verhielte es sich bei der Gruppierung von p_2 mit $(x_2 : T_2, x_3 : T_3)$. Die Konsequenz ist, dass der Algorithmus bei solchen Konflikten sich für eine der Möglichkeiten entscheiden muss. Die optimale Lösung hier wäre, die ungefähren Kosten für alle Möglichkeiten zu berechnen und anschließend die günstigste auszuwählen: dies setzt allerdings die Kenntnis der Größen der einzelnen Elementtypen und der Komplexität der Prädikate voraus und erfordert dementsprechend die Entwicklung von Kostenmodellen für die Auswertung von ODL-Termen, was weit über den Umfang der vorliegenden Arbeit hinausgeht. Dieser Lösungsansatz muss daher zukünftigen Entwicklungen vorbehalten bleiben.

Wir werden an dieser Stelle einen einfacheren Weg gehen, indem wir in solchen Konfliktsituationen dem zuerst verarbeiteten Term den Vorrang geben: im obigen Beispiel wäre es der Term $p_1(x_1, x_2)$. Dies ist ein Kompromiss zwischen dem Schwierigkeitgrad des Algorithmus und dem Optimierungsgrad des Ergebnisses.

Wir gehen nun wie folgt vor: alle Prädikate aus $M_2, \ldots, M_n$, die nach der Verarbeitung der Prädikate aus M_1 verblieben sind, werden nacheinander verarbeitet, und zwar aufsteigend geordnet nach der Anzahl der Argumente:

for $a = 2$ **to** n **do**
  for all $p_i \in M_a$ **do**
    if Es ist ein Elementtyp vorhanden, der die Elemente $x_{k_{i,1}}, \ldots, x_{k_{i,r_i}}$ des ursprünglichen Typs enthalt, die als Argumente von p_i verwendet werden (hier werden auch rekursiv Elementtypen von zusammengesetzten Elementtypen betrachtet) **then**
      Füge p_i in den Restriktionsterm des gefundenen Elementtyps ein und verknüpfe ihn mithilfe des logischen `AND` mit dem früheren Restriktionsterm.
    else
      co Kein Elementtyp enthält alle von p_i verwendeten Argumente **oc**
      Alle Elementtypen, die Argumente $x_{k_{i,j}}$ von p_i enthalten, werden zu einem neuen Elementtyp gruppiert. Als Restriktionsterm des neuen Elementtyps wird p_i benutzt.
    end if
  end for
end for

Nach dem Ende des Algorithmus ist der ursprüngliche eingeschränkte Typ so umformuliert, dass jedes Prädikat p_i aus seinem Restriktionsterm nach Möglichkeit mit den Elementtypen $x_{k_{i,1}}, \ldots, x_{k_{i,r_i}}$ gruppiert ist, die es als Argument verwendet.

Wir wollen die Arbeitsweise des Algorithmus an einem größeren Beispiel ausführlich erläutern. Es sei der eingeschränkte Typ

$$\{var : (x_1 : T_1, x_2 : T_2, x_3 : T_3, x_4 : T_4, x_5 : T_5) \mid \neg\Big(\big(p_1(x_1) \wedge p_2(x_4) \wedge p_3(x_1, x_2)\big) \Longrightarrow \Big(\big(p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3)\big) \Longrightarrow \big(\neg p_6(x_3, x_4) \vee \neg p_7(x_1, x_3, x_5)\big)\Big)\Big)\}$$

gegeben. Wie führen nun den Algorithmus schrittweise aus:

1: Äquivalenzen und Implikationen eliminieren:
Wir werden die Prädikate $p_1, \ldots, p_7$ bei den Äquivalenzumformungen der Kürze halber ohne Argumente notieren, weil diese keinen Einfluss auf die Umformungen haben:

$$\neg\Big(\big(p_1 \wedge p_2 \wedge p_3\big) \Longrightarrow \big((p_4 \wedge p_5) \Longrightarrow (\neg p_6 \vee \neg p_7)\big)\Big)$$
$$= \neg\Big(\big(p_1 \wedge p_2 \wedge p_3\big) \Longrightarrow \big(\neg(p_4 \wedge p_5) \vee (\neg p_6 \vee \neg p_7)\big)\Big)$$
$$= \neg\Big(\neg\big(p_1 \wedge p_2 \wedge p_3\big) \vee \big(\neg(p_4 \wedge p_5) \vee (\neg p_6 \vee \neg p_7)\big)\Big)$$

2: Eliminierung von Disjunktionen mithilfe von DeMorgans Gesetz:

$$\begin{aligned}
&\neg\Big(\neg(p_1 \wedge p_2 \wedge p_3) \vee \big(\neg(p_4 \wedge p_5) \vee (\neg p_6 \vee \neg p_7)\big)\Big)\\
=\ &\neg\Big(\neg(p_1 \wedge p_2 \wedge p_3) \vee \big(\neg(p_4 \wedge p_5) \vee \neg(p_6 \wedge p_7)\big)\Big)\\
=\ &\neg\Big(\neg(p_1 \wedge p_2 \wedge p_3) \vee \neg\big((p_4 \wedge p_5) \wedge (p_6 \wedge p_7)\big)\Big)\\
=\ &\Big((p_1 \wedge p_2 \wedge p_3) \wedge \big((p_4 \wedge p_5) \wedge (p_6 \wedge p_7)\big)\Big)\\
=\ &p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7
\end{aligned}$$

3: Überprüfung, dass keine ungeklammerten `OR`-Verknüpfungen vorkommen:
Im Term $p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7$ gibt es keine `OR`-Verknüpfungen, sodass der Algorithmus fortgesetzt werden kann.

4: Aufspaltung des Restriktionsterms in Teilterme:
Wir verteilen die durch das logische `AND` verknüpften Teilterme des Restriktionsterms auf die Mengen $M_1, \ldots, M_5$:

M_1 : $p_1(x_1),\ p_2(x_4)$

M_2 : $p_3(x_1, x_2),\ p_4(x_2, x_3),\ p_6(x_3, x_4)$

M_3 : $p_5(x_1, x_2, x_3),\ p_7(x_1, x_3, x_5)$

M_4 : leer, da keine Prädikate mit 4 Argumenten vorhanden

M_5 : leer, da keine Prädikate mit 5 Argumenten vorhanden

5: Teilterme mit den von ihnen verwendeten Elementtypen zu neuen Elementtypen gruppieren:

a) Zunächst werden die einstelligen Prädikate p_1 und p_2 aus M_1 mit ihren Argumenten zusammengeführt:

$$\{var : (x_1 : T_1, x_2 : T_2, x_3 : T_3, x_4 : T_4, x_5 : T_5) \mid p_1(x_1) \wedge p_2(x_4) \wedge p_3(x_1, x_2) \wedge p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3) \wedge p_6(x_3, x_4) \wedge p_7(x_1, x_3, x_5)\ \}$$

wird zu

$$\{var : (v_1 : \{x_1 : T_1 | p_1(x_1)\}, x_2 : T_2, x_3 : T_3, v_4 : \{x_4 : T_4 | p_2(x_4)\}, x_5 : T_5) \mid p_3(x_1, x_2) \wedge p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3) \wedge p_6(x_3, x_4) \wedge p_7(x_1, x_3, x_5)\ \}$$

umgeformt, wobei v_i neue lokale Variablen sind.

Um die Typdeklaration übersichtlich zu halten, kürzen wir einen Typ $v_i : \{x_i : T_i | p_j(x_i)\}$ mit $v_i : T_i p_j$ ab.

b) Jetzt verarbeiten wir die mehrstelligen Prädikate aus M_2 und M_3 (M_4 und M_5 sind leer und müssen nicht verarbeitet werden). Wir geben für jedes Prädikat aus M_2 und M_3 die Typdeklaration nach der Umformung, die dieses Prädikat mit seinen Argumenten gruppiert:

– $p_3(x_1, x_2)$: Die Elemente x_1 und x_2 werden zu einem Elementtyp mit dem Restriktionsterm $p_3(x_1, x_2)$ gruppiert:

$$\{var : (v_{12} : \{v_{12} : (v_1 : T_1 p_1, x_2 : T_2) | p_3(x_1, x_2)\}, x_3 : T_3, v_4 : T_4 p_2, x_5 : T_5) \mid p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3) \wedge p_6(x_3, x_4) \wedge p_7(x_1, x_3, x_5)\ \}$$

– $p_4(x_2, x_3)$: Die Elemente x_2 und x_3 müssen zu einem Elementtyp mit dem Restriktionsterm $p_4(x_2, x_3)$ gruppiert werden. Da x_2 bereits früher mit x_1 gruppiert wurde, ergibt sich ein neuer Elementtyp, der x_1, x_2 und x_3 gruppiert:

$$\{var : (v_{123} : \{v_{123} : (v_{12} : T_1 p_1 T_2 p_3, x_3 : T_3) | p_4(x_2, x_3)\}, v_4 : T_4 p_2, x_5 : T_5) \mid p_5(x_1, x_2, x_3) \wedge p_6(x_3, x_4) \wedge p_7(x_1, x_3, x_5)\ \}$$

– $p_6(x_3, x_4)$: Die Elemente x_3 und x_4 müssen zu einem Elementtyp gruppiert werden. Da x_3 bereits in einem Elementtyp mit x_1 und x_2 zusammengefasst wurde, muss x_4 jetzt mit diesem Elementtyp gruppiert werden:

$\{var : (v_{1234} : \{v_{1234} : (v_{123} : T_1 p_1 T_2 p_3 T_3 p_4, x_4 : T_4) | p_6(x_3, x_4)\}, x_5 : T_5) \mid$
$p_5(x_1, x_2, x_3) \wedge p_7(x_1, x_3, x_5) \}$

– $p_5(x_1, x_2, x_3)$: Hier tritt der Fall ein, wo alle Argumente des Prädikats bereits in einem Elementtyp zusammengefasst sind, und zwar im Elementtyp, dessen Variable var_{123} heißt. Deshalb wird p_5 zum Restriktionsterm von var_{123} mit einem logischen AND hinzugefügt:
  $\{var : (v_{1234} : \{v_{1234} : (v_{123} : \{v_{123} : (v_{12} : T_1 p_1 T_2 p_3, x_3 : T_3) | p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3)\},$
  $x_4 : T_4) | p_6(x_3, x_4)\}, x_5 : T_5) \mid p_7(x_1, x_3, x_5) \}$

– $p_7(x_1, x_3, x_5)$: Für diesen Term kann keine Optimierung mehr vorgenommen werden, da die Gruppierung von x_1, x_3 und x_5 zu einem Elementtyp die Zusammenfassung der verbliebenen Typelemente var_{1234} und x_5 nach sich ziehen würde – das Ergebnis wäre der Typ $(var_{1234} : T_1 p_1 T_2 p_3 T_3 p_4 p_5 T_4 p_6, x_5 : T_5)$, der schon jetzt der Basistyp des äußeren eingeschränkten Typs ist.

Nachdem wir den Ablauf der Optimierung dargestellt haben, wollen wir die optimierte Formulierung des als Beispiel verwendeten eingeschränkten Typs angeben:
$\{var : (v_{1234} : \{v_{1234} : (v_{123} : \{v_{123} : (v_{12} : \{v_{12} : (x_1 : \{x_1 : T_1 | p_1(x_1)\}, x_2 : T_2) | p_3(x_1, x_2)\}, x_3 : T_3) |$
$p_4(x_2, x_3) \wedge p_5(x_1, x_2, x_3)\}, x_4 : T_4) | p_6(x_3, x_4)\}, x_5 : T_5) \mid p_7(x_1, x_3, x_5)\}$
Diese Formulierung bringt unter Umständen erhebliche Zeitgewinne bei der Auswertung gegenüber der äquivalenten nicht-optimierten Formulierung
$\{var : (x_1 : T_1, x_2 : T_2, x_3 : T_3, x_4 : T_4, x_5 : T_5) \mid p_1(x_1) \wedge p_2(x_4) \wedge p_3(x_1, x_2) \wedge p_4(x_2, x_3) \wedge$
$p_5(x_1, x_2, x_3) \wedge p_6(x_3, x_4) \wedge p_7(x_1, x_3, x_5)\}$
ist jedoch gleichzeitig viel unübersichtlicher. Genau das ist auch der Grund dafür, dass die Integration dieser Optimierung in das ODL-Auswertungssystem von großem Vorteil wäre, denn es ist einem Benutzer kaum zuzumuten, ODL-Abfragen selbst so weitgehend zu optimieren, dass sie wie die obige optimierte Typdeklaration aussehen.

Bei der Implementierung ist folgender Aspekt zu beachten: die ODL-Notation schreibt, im Unterschied zu der im Algorithmus verwendeten vereinfachten Notation, die Verwendung von Selektoren zum Zugriff auf Elemente des Basistyps eines eingeschränkten Typs vor: die korrekte Schreibweise für den Zugriff auf die Basistyp-Elemente in unserem Beispieltyp wäre also

```
{ var:(x1:T1,x2:T2,x3:T3,x4:T4,x5:T5) |
  p1(var.x1) and p2(var.x4) and p3(var.x1,var.x2) and
  p4(var.x2,var.x3) and p5(var.x1,var.x2,var.x3) and
  p6(var.x3,var.x4) and p7(var.x1,var.x3,var.x5) }
```

In der optimierten Formulierung müsste beispielsweise das Prädikat p_7 im Restriktionsterm sogar kompliziertere Selektorausdrücke verwenden:

```
p7(var.v1234.v123.v12.x1, var.v1234.v123.x3, var.x5)
```

Wie sich damit zeigt, entsteht bei der Umformulierung eines eingeschränkten Typs das Problem, dass die Namen der Basistyp-Elemente verändert werden, sodass für den Zugriff sowohl im Restriktionsterm als auch in Termen außerhalb des eingeschränkten Typs (das sind Terme, welche die Variable benutzen, deren Typ optimiert wurde) angepasste Selektoren benutzt werden müssten. Ferner fiele auch die Darstellung der erfüllenden Belegungen für die Variable, deren Typ optimiert wurde, anders aus, als der Benutzer sie gemäß der ursprünglichen Typdeklaration erwarten würde.

Eine Möglichkeit, diesen Problemen aus dem Weg zu gehen, besteht darin, dass die Umformulierung eines eingeschränkten Type innerhalb der ODL-Auswertungsklasse `MetaRestrictedType` gekapselt wird, die den eingeschränkten Typ darstellt (s. auch Abschnitt 5.1.2). Der optimierte eingeschränkte Typ verhielte sich nach außen wie seine nicht-optimierte Version – insbesondere könnte auf die Basistyp-Elemente mit denselben Selektoren wie bei der Ausgangsformulierung des Typs zugegriffen werden – für die Iteration über die Typinstanzen würde aber intern die optimierte Formulierung verwendet. Weiter könnte die gekapselte optimierte Formulierung zur Überprü-

fung eingesetzt werden, ob ein Tupel, bei dem einige der Elementwerte nicht festgelegt sind, die Restriktionsbedingung verletzt: die optimierte Version des obigen Beispieltyps könnte für ein Tupel `(x1=Wert1,x2=Wert2,x3=Wert3)` feststellen, ob es die Restriktionsbedingung verletzt, da jedes Tupel, das die gesamte Restriktionsbedingung erfüllt, auch die Restriktionsbedingung von v_{123} erfüllten müsste – diese Funktionalität wäre für Eingabedialoge für eingeschränkte Typen nützlich, da der Benutzer auf diese Weise früher informiert werden könnte, wenn die von ihm eingegebenen Werte die Restriktionsbedingung nicht erfüllen.

Zum Schluss wollen wir noch einige Weiterentwicklungsmöglichkeiten für die beschriebene Optimierung nennen:

- **Anwendung eines Kostenmodells bei der Optimierung**:
  Wie bei der Beschreibung des Algorithmus im Schritt 5b auf der Seite 107 erwähnt, kann für Situationen, in denen es mehrere Möglichkeiten gibt, Elemente des Basistyps des zu optimierenden eingeschränkten Typs mit Prädikaten aus dem Restriktionsterm zu gruppieren, eine Berechnung der Kosten für die Auswertung jeder Gruppierungsmöglichkeit vorgenommen werden und dann diejenige Optimierung durchgeführt werden, die zu den geringsten Auswertungskosten führt.

- **Optimierung von Quantortermen**:
  Die vorgestellte Optimierung kann auf von Quantoren gebundene Terme ausgedehnt werden, indem der Typ der vom Quantor gebundenen Variablen zu einem eingeschränkten Typ umgewandelt wird, der als Restriktionsterm den Quantorenterm oder einige Teilterme aus ihm enthält. Anschließend kann der Optimierungsalgorithmus auf den Restriktionsterm angewandt werden.

  Betrachten wir die Beispielabfrage
  ```
  exists ports:( p1:Port, p2:Port ).(
    ports.p1.Name = ”Slot1” and ports.p2.Name = ”Slot” and
    exists ch:Channel.(
      ch.SourcePort = ports.p1 and
      ch.DestinationPort = ports.p2 ) )
  ```
  Da der Quantorterm eine Konjunktion mehrerer Teilterme darstellt, können wir einige Teilterme in den Typ der Quantorvariablen verlagern, der dafür zu einem eingeschränkten Typ umgewandelt werden muss:
  ```
  exists ports:{ p:( p1:Port, p2:Port ) |
    p.p1.Name = ”Slot1” and p.p2.Name = ”Slot” }.
    exists ch:Channel.(
      ch.SourcePort = ports.p1 and
      ch.DestinationPort = ports.p2 )
  ```
  Bereits diese Umformung bringt Zeitgewinne bei der Auswertung. Nach der Anwendung des Optimierungsalgorithmus auf den eingeschränkten Typ erhalten wir eine Formulierung der Abfrage, bei der die Auswertungszeit noch einmal sinkt:
  ```
  exists ports:(
    p1:{ p1:Port | p1.Name = ”Slot1” },
    p2:{ p2:Port | p2.Name = ”Slot” } ).
    exists ch:Channel.(
      ch.SourcePort = ports.p1 and
      ch.DestinationPort = ports.p2 )
  ```

6.3 Verbesserungen an der Benutzerschnittstelle

Dieser Abschnitt beschäftigt sich mit den Verbesserungen an der interaktiven Benutzerschnittstelle von ODL.

6.3.1 Konfiguration der Eingabedialoge während der Eingabe

Die Konfiguration der Eingabedialoge für verschiedenen ODL-Datentypen findet in einem separaten Konfigurationsdialog statt (s. Abschnitt 4.2.7), in dem für jeden ODL-Datentyp (außer dem eingeschränkten Typ) eingestellt werden kann, welcher Eingabebereich und welches Eingabedialog-Fenster für diesen Typ verwendet werden soll.

In der aktuellen Implementierung ist keine Möglichkeit vorgesehen, die Eingabedialog-Einstellungen während der Ausführung einer ODL-Abfrage zu ändern: die Eingabedialoge sind modal, sodass der Konfigurationsdialog während der Eingabe nicht gestartet werden kann. Für den Benutzer wäre es jedoch von Vorteil, wenn ein Eingabedialog während der laufenden Eingabe an seine Bedürfnisse angepasst werden könnte.

Der modulare Aufbau des ODL-Query-Subsystems lässt eine solche Erweiterung ohne größeren Aufwand zu. Hierfür müsste eine Möglichkeit geschaffen werden, den Konfigurationsdialog aus einem Eingabedialog heraus aufzurufen, beispielsweise mithilfe eines zusätzlichen *Options*-Buttons im Eingabedialog. Die vom Benutzer vorgenommenen Einstellungen würden dann sofort für die laufende Eingabe angewandt, wobei je nach Änderungen der Einstellungen der Eingabebereich oder der gesamte Eingabedialog neu initialisiert und angezeigt werden müsste.

6.3.2 Verbesserungen bei der Eingabe eingeschränkter Typen

Eingaben von Werten eingeschränkter Typen, die bei der Auswertung von ODL-Abfragen der Form

```
context var:{ localVar:base_type | restriction_term}
```

ausgeführt werden, stellen oft eine Schwierigkeit für den Benutzer dar, weil hier, im Unterschied zu allen anderen ODL-Datentypen, nicht jeder gültige Basistyp-Wert auch eine zulässige Eingabe darstellt, wie das bei anderen Typen ist. Diese Besonderheit ist stellt kein größeres Problem dar, solange der Basistyp des eingeschränkten Typs unär und endlich ist, denn in diesem Fall wird die Eingabe durch die Auswahl eines Werts aus der Liste durchgeführt, und hier kann der Benutzer Einträge aus der Liste nacheinander selektieren, bis ein zulässiger Wert gefunden und der *Next*-Button des Eingabedialogs aktiviert wird (eine Ausnahme bildet der Typ `Boolean`, dessen Werte auch in einem Textfeld eingegeben werden können, was aber wiederum kein Problem darstellt, weil es nur zwei verschiedene boolesche Werte gibt, sodass der Benutzer durch Probieren schnell herausfinden kann, welche Werte die Restriktionsbedingung erfüllen). Wird jedoch ein eingeschränkter Typ abgefragt, dessen Basistyp zusammengesetzt ist (beispielsweise ein Produkttyp), so ist es für den Benutzer oft schwierig festzustellen, welcher der Elementwerte des Basistyps die Restriktionsbedingung verletzt bzw. wie eine zulässige Kombination von Elementwerten des Basistyps aussehen soll.

In diesem Abschnitt besprechen wir verschiedene Möglichkeiten, die Eingabe von Werten eingeschränkter Typen für den Benutzer einfacher zu gestalten. Die meisten davon verwenden eine Vorausfilterung der einzugebenden Werte, d.h., dem Benutzer werden nur diejenigen Werte zur Auswahl angeboten, die die Restriktionsbedingung erfüllen. Bei dieser Filterung gibt mehrere Realisierungsmöglichkeiten, die sich im Schwierigkeitsgrad und dem sich für den Benutzer ergebenden Komfort unterscheiden. Wir wollen im Folgenden einige Möglichkeiten beschreiben. Ihre Reihenfolge entspricht dabei dem zu erwartenden Schwierigkeitsgrad der Implementierung – wir beginnen mit der einfachsten und beenden mit der schwierigsten Lösung.

1) **Abfrage zusammengesetzter Typen, die eingeschränkte Typen als Elemente enthalten:**
   Bei der Eingabe eines eingeschränkten Typs ist der Abschluss der Eingabe erst dann möglich,

wenn der eingegebene Wert des Basistyps die Restriktionsbedingung erfüllt. Der *Next*-Button wird demnach genau dann aktiviert, wenn die eingegebenen Werte die Restriktionsbedingung erfüllen. Bei einem eingeschränkten Typ mit unärem Basistyp kann damit, wie oben bereits angesprochen, am *Next*-Button direkt abgelesen werden, ob ein korrekter Wert eingegeben wurde. Handelt es sich bei dem Basistyp jedoch um einen zusammengesetzten Typ, so kann es für den Benutzer schwierig sein herauszufinden, welche Kombinationen der Elementwerte die Restriktionsbedingung erfüllen.

Einen Spezialfall stellen zusammengesetzte Typen dar, deren Elementtypen eingeschränkte Typen sind. Hier ist zwar, wie im Allgemeinen bei eingeschränkten Typen, an einem deaktivierten *Next*-Button zu erkennen, ob die eingegebenen Werte die Restriktionsbedingungen der Elementtypen verletzen, nicht jedoch, welche Einzelwerte zu korrigieren sind. Nehmen wir als Beispiel die Abfrage:

```
context [ hint = ”Enter an existing component name and
                  an existing port name” ] names:(
  compName:{ s:String | exists c:Component. c.Name = s },
  portName:{ s:String | exists p:Port. p.Name = s } ). true
```

Hier muss der Benuzter den Namen einer im Modell existierenden Komponente und den Namen eines existierenden Ports eingeben. Wie der Abbildung 6.1 zu entnehmen ist, kann der Benutzer nicht direkt sehen, welcher der zwei einzugebenden Werte seine Restriktionsbedingung verletzt.

Die Information darüber, welche der eingegebenen Elementwerte ihre Restriktionsbedingung erfüllen, kann dem Benutzer zur Verfügung gestellt werden, indem zu jedem `RestrictedType`-Wert im Eingabebereich angezeigt wird, ob der eingegebene Wert zulässig ist. Dies kann beispielsweise mithilfe einer Checkbox geschehen, wie auf der mit einem Graphikeditor bearbeiteten Abbildung 6.2 dargestellt.

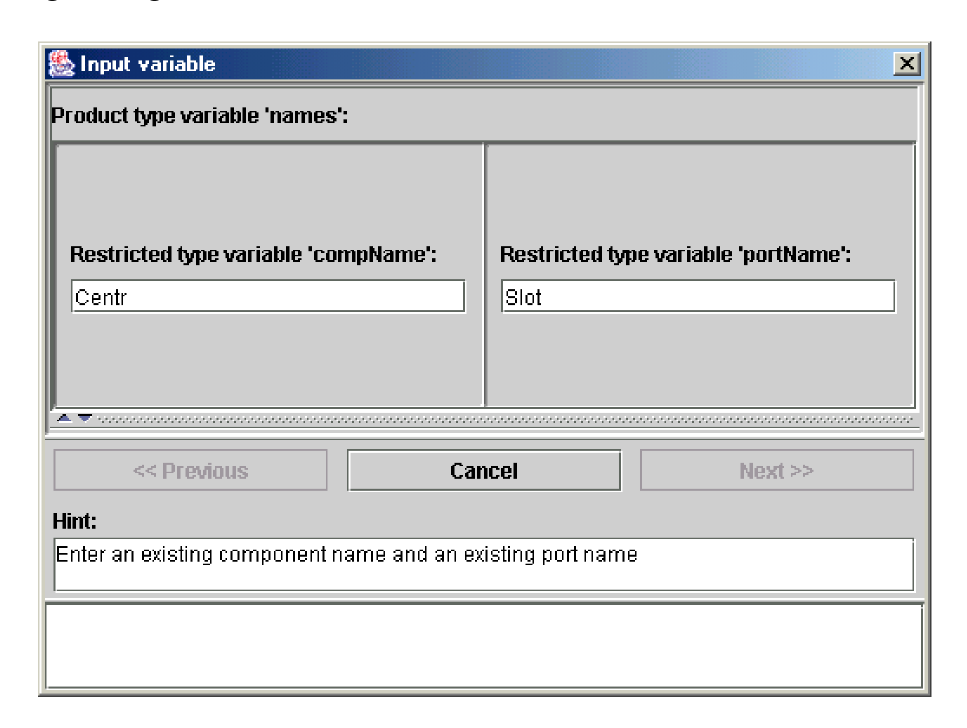


Abbildung 6.1: Eingabe von `RestrictedType`-Werten

Diese Verbesserung ließe sich mit vergleichsweise geringem Aufwand realisieren. Sie bietet allerdings keine umfassende Lösung für die Eingabe eingeschränkter Typen, sondern lediglich eine Erleichterung bei der Eingabe zusammengesetzter Typen, die eingeschränkte Typen als Elemente enthalten.

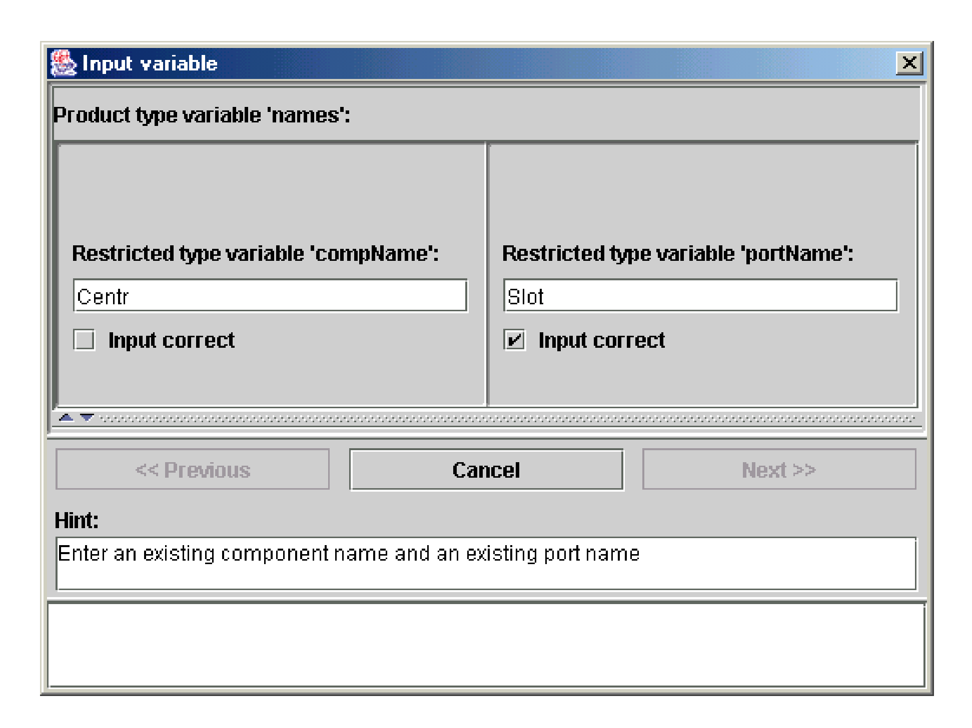


Abbildung 6.2: Eingabe von `RestrictedType`-Werten mit Korrektheitsanzeige

2) **Auflistung aller zulässigen Werte des eingeschränkten Typs:**
Eine vergleichsweise einfach zu implementierende allgemeine Lösung für die Eingabe eingeschränkter Typen ist, sämtliche zulässigen Werte im Eingabebereich aufzulisten, damit der Benutzer einen der Werte auswählen kann. Die Abbildung 6.3, die mithilfe eines Graphikeditors erstellt wurde, zeigt, wie ein solcher Eingabedialog für einen eingeschränkten Typ mit dem Basistyp

```
(c1:Component, c2:Component, c3:Component)
```

aussehen könnte.

Der Vorteil dieser Lösung gegenüber der aktuellen Implementierung ist, dass jeder Wert in der Auflistung die Restriktionsbedingung erfüllt und damit auch eingegeben werden darf, sodass der Benutzer nicht mehr selbst auf die Einhaltung der Restriktionsbedingung achten muss. Die Auflistung aller zulässigen Werte hat aber auch Nachteile:

- Zum Aufbau der Liste müssen alle Basistypwerte auf die Erfüllung der Restriktionsbedingung überprüft werden. Dies kann für große Basistypen, insbesondere für Produkttypen, längere Zeit in Anspruch nehmen, sodass der Benutzer warten muss, bis der Eingabedialog erscheint.
- Wenn viele Basistypwerte die Restriktionsbedingung erfüllen, kann die Auflistung sehr lang werden: mehrere Zehntausend Einträge sind durchaus denkbar. Bereits die relativ einfache ODL-Abfrage

  ```
  context ports:{ p:(p1:Port, p2:Port) |
    neg p.p1 = p.p2 }.true
  ```

  bei der zwei verschiedene Ports ausgewählt werden müssen, kann einige tausend Einträge in der Auflistung erzeugen. Dies erschwert zum Einen die Suche nach dem gewünschten Eintrag und kann zum Anderen die Ressourcen des verwendeten Rechners überstrapazieren: eine Abfrage, bei der nicht eines, sondern drei Paare verschiedener Ports auszuwählen sind, würde eine Liste aus mehreren Milliarden Einträgen erfordern, deren Anzeige nicht möglich sein dürfte.

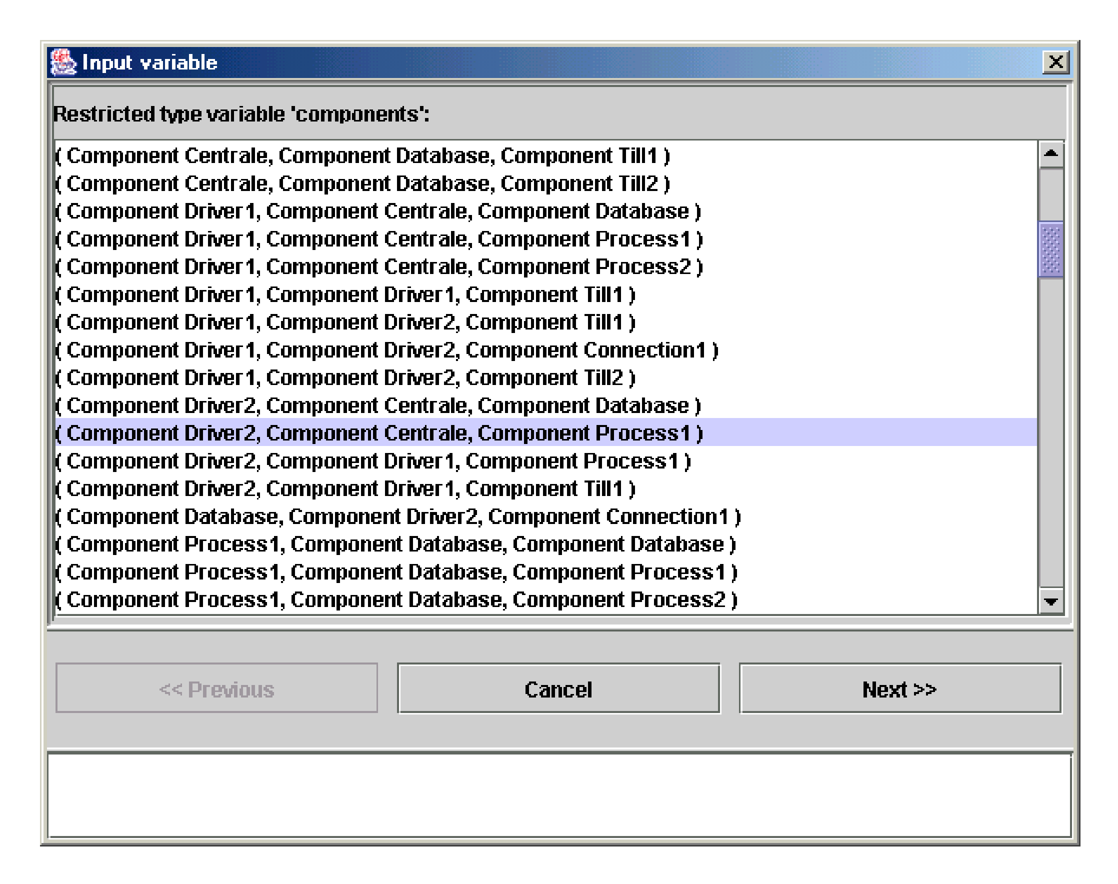


Abbildung 6.3: Auflistung aller zulässigen `RestrictedType`-Werte im Eingabebereich

Angesichts der oben aufgeführten Nachteile kann festgehalten werden, dass die Eingabe eines `RestrictedType`-Werts über die Auswahl aus einer vollständigen Auflistung nur dann sinnvoll ist, wenn der Basistyp des eingeschränkten Typs nicht zu groß ist (höchstens einige zehntausend Werte) und die Anzahl der zulässigen Werte, welche die Restriktionsbedingung erfüllen, einige hundert nicht übersteigt. Diese Lösung kann daher nur als zusätzliche Eingabemöglichkeit neben einer existierenden Lösung implementiert werden, die vom Benutzer im Eingabedialog während der Eingabe eingeschaltet und wieder abgeschaltet werden kann.

3) Filterung des letzten auszuwählenden Elements eines *n*-elementigen eingeschränkten Typs: Im Rahmen von [Tracht] wurde zur Auswahl von Portpaaren gleichen Datentyps ein Auswahlmechanismus implementiert, bei dem die Selektion eines Ports aus der einen Liste zur Filterung der zweiten Liste führte, sodass sie nur noch Ports zeigte, die denselben Datentyp wie der ausgewählte Port in der ersten Liste haben.

Dieses Prinzip kann für die Eingabe eingeschränkter Typen in ODL-Abfragen verallgemeinert werden:

> Ist ein Wert für einen eingeschränkten Typ einzugeben, dessen Basistyp ein zusammengesetzter Typ mit n Elementen ist, so kann nach der Eingabe von $n-1$ Elementen das n-te Element, sofern es über eine Auswahl und nicht in einem Textfeld einzugeben ist, so gefiltert werden, dass nur Elemente zur Auswahl stehen, die zusammen mit den bereits eingegebenen Werten für die anderen $n-1$ Elemente die Restriktionsbedingung erfüllen. Für den Spezialfall $n = 1$, d.h. für unäre Typen und für zusammengesetzte Typen mit genau einem Element, wird die Filterung des einzigen Elements des Basistyps sofort beim Öffnen des Eingabedialogs durchgeführt.

Betrachten wir als Beispiel die ODL-Abfrage

```
context comps:{ c:( c1:Component, c2:Component ) |
  c.c2.SuperComponent = c.c1 }.true
```
(6.9)

Der Benutzer muss zwei Komponenten auswählen, wobei die zweite eine Unterkomponente der ersten sein muss. Die Filterung würde dazu führen, dass bei der Auswahl einer Komponente für `c1` die Liste für `c2` gefiltert wird und anschließend nur die Unterkomponenten von `c1` anzeigt. Analog führt die Selektion einer Komponente für `c2` dazu, dass die Liste für `c1` nur die Oberkomponente von `c2`, falls vorhanden, aufführt. Auf der in einem Graphikeditor nachbearbeiteten Abbildung 6.4 wird gezeigt, wie der Eingabedialog für die obige Abfrage aussehen könnte, nachdem eine Komponente für `c1` ausgewählt wurde.

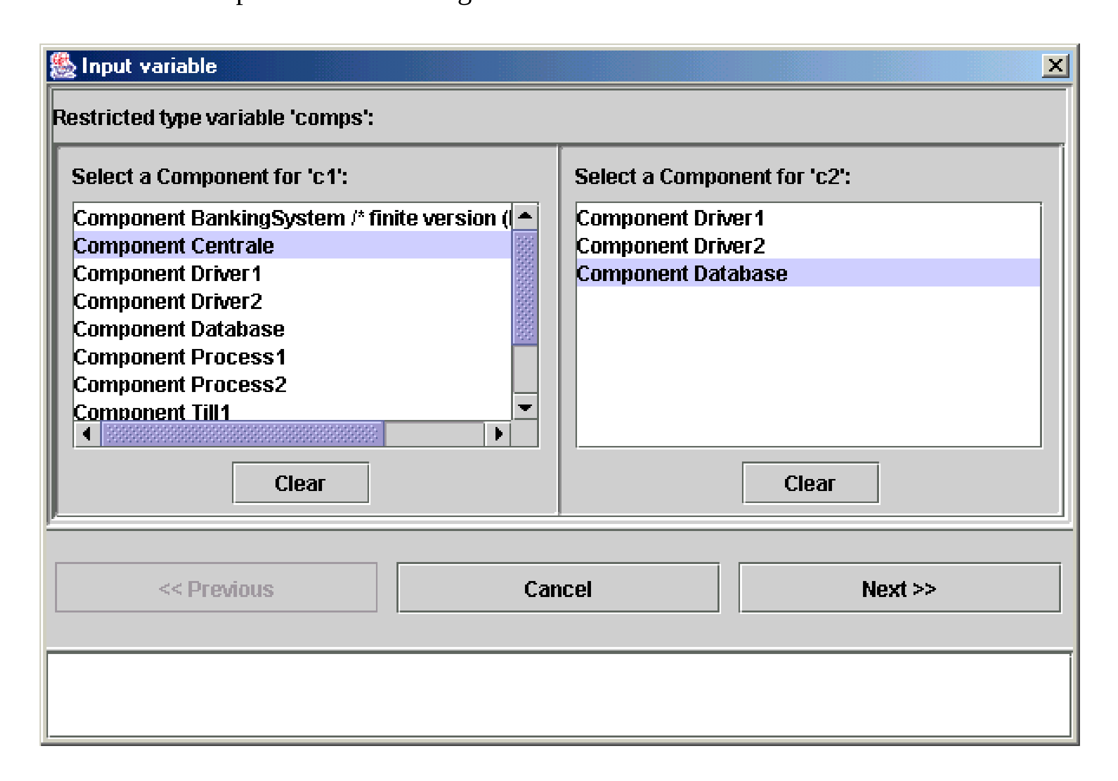


Abbildung 6.4: Filterung des letzten auszuwählenden Basistyp-Elements eines eingeschränkten Typs

Bei der Implementierung der Filterung ist darauf zu achten, dass eine Liste, in der bereits ein Wert ausgewählt ist, nicht mehr gefiltert werden soll, wenn der Benutzer in einer anderen Liste einen Wert auswählt. Wird beispielsweise in der obigen Abfrage eine Komponente für `c1` und anschließend eine für `c2` ausgewählt, so darf die Liste für `c1` nicht mehr gefiltert werden, weil ihre Auswahl vor der Auswahl in der zweiten Liste vorgenommen wurde und damit ihr gegenüber priorisiert ist. Diese Priorität wird aufgehoben, wenn der Benutzer die Auswahl in der ersten Liste mithilfe des zu ihr gehörenden *Clear*-Buttons aufhebt.

Die beschriebene Optimierung erfordert einen höheren Implementierungsaufwand als die Optimierungsvorschläge 1) und 2), weil hier mehrere Änderungen an ODL-Query-Subsystem vorgenommen werden müssen:

- Auswahllisten müssen ihre Werte nicht direkt von den ODL-Datentypen, sondern über eine Zwischenliste beziehen, auf die Filter angewandt werden können.
- In allen Eingabepanels, bei denen die Eingabe durch die Auswahl eines Werts aus einer Auflistung aller verfügbaren Werte stattfindet, muss die Möglichkeit geschaffen werden, eine Kontrollklasse für die Filterung der Auflistung anzuschließen.

- Es muss eine Kontrollklasse erstellt werden, die die Filterung von Elementwerten in Eingabepanels eingeschränkter Typen entsprechend dem beschriebenen Optimiertungsvorschlag steuert.

Die Implementierung dieser Optimierung würde die Eingabe eingeschränkter Typen für den Benutzer beträchtlich erleichtern, insbesondere bei eingeschränkten Typen, deren Basistypen aus wenigen Elementen bestehen. Ein weiterer Vorteil dieser Lösung ist, dass sich das Erscheinungsbild des Eingabepanels gegenüber der aktuellen Implementierung, im Gegensatz zum Vorschlag 2), nur unwesentlich verändert würde. Es ist jedoch beachten, dass die Filterung bei komplexen Restriktionstermen viel Zeit beanspruchen kann, sodass dem Benutzer eine Möglichkeit zur Verfügung gestellt werden muss, die Filterung zu deaktivieren – beispielsweise durch eine Checkbox *"Filter last value"*, die während der Eingabe gesetzt oder gelöscht werden kann.

4) Filterung aller Elemente eines eingeschränkten Typs:
Eine Weiterentwicklung des Optimierungsvorschlags 3) besteht darin, dass nicht nur der letzte, sondern alle Elemente des Basistyps eines eingeschränkten Typs vor und während der Eingabe gefiltert werden. Das Grundprinzip kann folgendermaßen formuliert werden:

> Ist ein Wert für einen eingeschränkten Typ einzugeben, dessen Basistyp ein zusammengesetzter Typ mit n Elementen ist, so können nach der Eingabe von $n-m$ Elementen ($m < n$) die restlichen m Elemente so gefiltert werden, dass nur Elemente zur Auswahl stehen, die zusammen mit den bereits eingegebenen Werten für die anderen $n-m$ Elemente in mindestens einem Basistyp-Tupel auftauchen, das die Restriktionsbedingung erfüllt.

Wir wollen die Idee für einen Algorithmus beschreiben, der diese Optimierung realisiert:

- Vor dem Öffnen des Eingabedialogs wird über alle Instanzen des eingeschränkten Typs iteriert. Für jede solche Instanze, d.h., für jedes Basistyp-Tupel, das die Restriktionsbedingung erfüllt, werden die Elemente $x_1, \ldots, x_n$ in Hashtabellen $H_1, \ldots, H_n$ eingetragen – wenn ein Element x_i bereits in der Tabelle H_i vorhanden ist, muss es nicht erneut eingetragen werden.

  Nach der Iteration über alle Instanzen des eingeschränkten Typs befinden sich in den Tabellen $H_1, \ldots, H_n$ alle Werte der Basistyp-Elemente, die in mindestens einem Tupel vorkommen, das die Restriktionsbedingung erfüllt. Die Werte der Basistyp-Elemente aus $H_1, \ldots, H_n$ werden nun in den Auswahllisten für die Elemente $x_1, \ldots, x_n$ des Basistyps angezeigt.
- Wird der Wert für ein erstes Basistyp-Element x_{i_1} ausgewählt, so müssen die restlichen $n-1$ Auswahllisten gefiltert werden. Dafür wird über alle Basistyp-Tupel iteriert, deren Elementwerte $x_1, \ldots, x_n$ in den Tabellen $H_1, \ldots, H_n$ enthalten sind, und deren i_1-tes Element auf den ausgewählten Wert von x_{i_1} gesetzt ist. Die Werte von Elementen $x_1, \ldots, x_{i_1-1}, x_{i_1+1}, x_n$, die in mindestens einem Tupel vorkamen, das die Restriktionsbedingung erfüllte, werden in den Listen für die Elemente $x_1, \ldots, x_{i_1-1}, x_{i_1+1}, x_n$ angezeigt.
- Wird der Wert für ein Basistyp-Element x_{i_r} ausgewählt $(1 \leq r \leq n-1)$, nachdem Werte für die Elemente $x_{i_1}, \ldots, x_{i_{r-1}}$ ausgewählt wurden, so werden die restlichen $n-r$ Elemente x_j mit $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_r\}$ analog zum vorherigen Schritt gefiltert: in den Auswahllisten für die Elemente x_j werden nur diejenigen Werte angezeigt, die in mindestens einem Basistyp-Tupel vorkommen, das die Restriktionsbedingung erfüllt, und in dem die Elemente $x_{i_1}, \ldots, x_{i_r}$ auf die ausgewählten Werte gesetzt sind.
- Wird bei einem früher ausgewählten Element x_{i_k} die Selektion aufgehoben, dann wird so vorgegangen, als wäre das zuletzt ausgewählte Element x_{i_r} erneut ausgewählt worden, wobei der Wert für x_{i_k} nicht mehr fest ist, sondern ebenfalls gefiltert wird: die ausgewählten Elemente sind nun $x_{i_1}, \ldots, x_{i_{k-1}}, x_{i_{k+1}}, \ldots, x_{i_r}$ und die gefilterten Elemente sind x_j mit $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_r\} \bigcup \{i_k\}$.

Nach der Implementierung dieser Optimierung könnte der Eingabedialog für die Beispielabfrage 6.9 wie auf der mit einem Graphikeditor bearbeiteten Abbildung 6.5 aussehen. Im Unterschied zu dem Eingabedialog, das bei der Filterung des letzten auszuwählenden Elements angezeigt wird (Abbildung 6.4), führt die Liste für das Basistyp-Element `c1` nur diejenigen Komponenten auf, die Unterkomponenten besitzen und daher in Basistyp-Tupeln auftauchen, die die Restriktionsbedingung erfüllen.

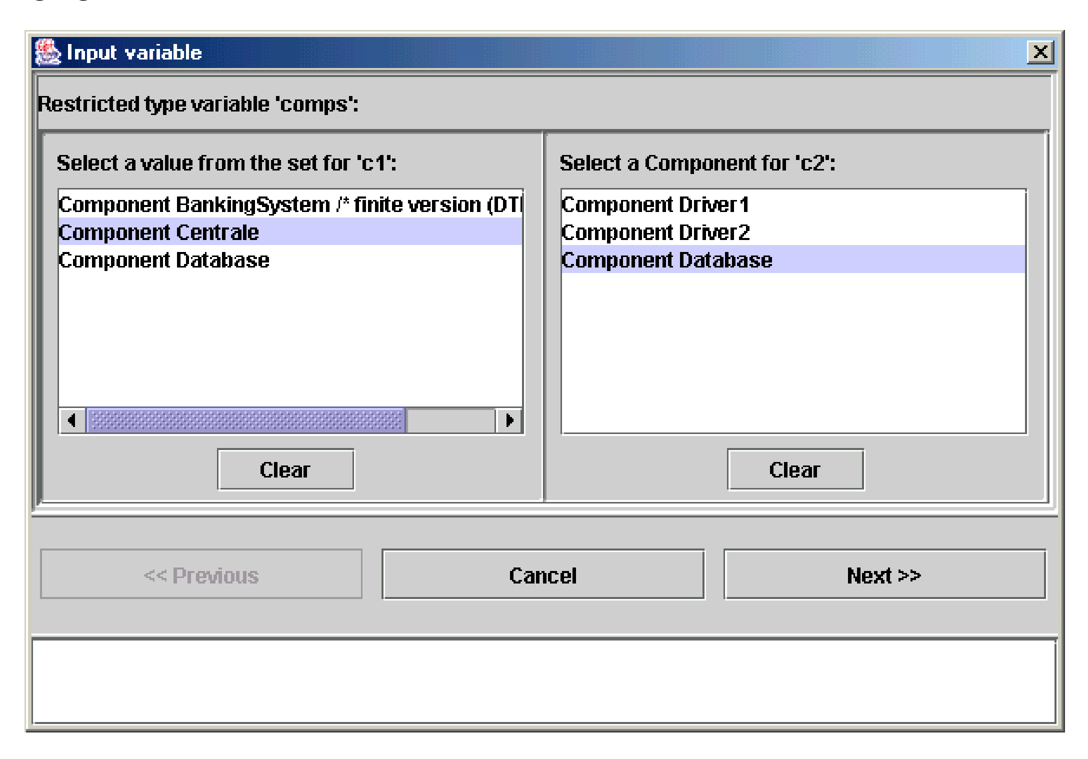


Abbildung 6.5: Filterung aller Basistyp-Elemente eines eingeschränkten Typs

Dieser Verbesserungsvorschlag bietet mehrere Vorteile:

- Der große Vorteil des hier beschriebenen Verbesserungsvorschlags ist, dass der Benutzer bei der Eingabe von Basistyp-Werten für einen eingeschränkten Typ nur solche Elementwerte auswählen kann, für die eine Auswahl der restlichen Elementwerte existiert, die zusammen mit den bereits ausgewählten Elementwerten eine zulässige Eingabe für den eingeschränkten Typ ergeben können. Anders ausgedrückt, wenn Werte für die Elemente $x_{i_1}, \ldots, x_{i_r}$ bereits ausgewählt sind, so ist jeder der Werte für x_j mit $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_r\}$, der in der entsprechenden j-ten Auswahlliste angezeigt wird, in mindestens einem Basistyp-Tupel enthalten, das die Restriktionsbedingung des eingeschränkten Typs erfüllt.
- Wie schon beim Vorschlag 3) muss das Erscheinungsbild des Eingabepanels gegenüber der aktuellen Implementierung nur unwesentlich verändert werden. Insbesondere muss dem Benutzer eine Möglichkeit zur Verfügung gestellt werden, die Filterung während der Eingabe zu aktivieren oder zu deaktivieren, indem er beispielsweise eine Checkbox *"Filter values"* setzt oder löscht.

Schließlich müssen wir auf die Probleme bei diesem Verbesserungsvorschlags eingehen:

- Der Implementierungsaufwand ist höher als bei allen vorher vorgestellten Verbesserungsvorschlägen für die Eingabe eingeschränkter Typen.
- Besteht der Basistyp aus vielen Elementen und/oder ist der Restriktionsterm aufwändig auszuwerten, so kann die Filterung vor dem Öffnen des Eingabedialogs viel Zeit in Anspruch nehmen.

- Nach jeder Auswahl eines Werts müssen alle Listen, in denen kein Wert ausgewählt ist, gefiltert werden, was ebenfalls eine beträchtliche Zeit beanspruchen kann, wenngleich nicht mehr als die Zeit für die initiale Filterung vor dem Öffnen des Eingabedialogs.

Wie wir sehen, hängen die Nachteile, die zur Laufzeit während der Benutzereingaben zutage treten, mit der hohen Rechenintensität der Filterung zusammen. Deshalb könnte zur Senkung des Zeitbedarfs bei der Filterung auf den Optimierungsalgorithmus für die Auswertung eingeschränkter Typen zurückgegriffen werden, der im Abschnitt 6.2.2 vorgestellt wurde.

5) **Kombination aus mehreren Verbesserungsvorschlägen:**
Die Verbesserungsvorschläge 1), 2) und 4) können gleichzeitig implementiert werden (dabei ist 3) ein Spezialfall von 4), sodass nur eine der beiden Möglichkeiten implementiert werden sollte). Es würde deshalb Sinn machen, mehrere Optimierungen zu realisieren und dann dem Benutzer die Auswahl zwischen konkurrierenden Ansätzen – hier sind es nur 2) und 4) – zu ermöglichen. Der Benutzer könnte also, abhängig von dem aktuell abgefragten eingeschränkten Typ, im Eingabedialog auswählen, ob alle Basistyp-Werte, welche die Restriktionsbedingung erfüllen, in einer Liste aufgeführt werden sollen, oder ob einzelne Auswahllisten für die Basistyp-Elemente angezeigt und gefiltert werden sollen.

6.4 Konzept einer flexiblen Dialogflusskontrolle

In diesem Abschnitt werden verschiedene Möglichkeiten zur Erweiterung und Verbesserung der Dialogflusskontrolle bei Benutzereingaben diskutiert.

Zunächst wollen wir die Motivation für die Weiterentwicklung der Dialogflusskontrolle anhand eines Beispiels erklären. Im Rahmen meines Systementwicklungsprojekts wurde eine interaktive Modelltransformation realisiert, bei der eine Komponente in ein Kanalbündel eingefügt werden musste, wobei die Kanäle durch die eingefügte Komponente "durchgeleitet" wurden, d.h., die Kanäle wurden gelöscht und ihre Ports durch neue Kanäle mit den Ports der eingefügten Komponente verbunden. Um den Vorgang zu starten, muss der Benutzer eine Hauptkomponente auswählen, deren Kanäle und Unterkomponenten an der Transformation teilnehmen werden, und anschließend den Menüpunkt `"Edit → Split channels"` auswählen. Es beginnt ein dialogbasierter Vorgang, bei dem der Benutzer in drei aufeinander folgenden Dialogen folgende Eingaben machen soll:

1: Auswahl mehrerer Kanäle, die das Kanalbündel bilden.

2: Auswahl der Komponente, die in das Kanalbündel einzufügen ist.

3: Zuordnung der Ausgangsports des Kanalbündels zu Eingangsports der eingefügten Komponente und Zuordnung der Eingangsports des Kanalbündels zu Ausgangsports der eingefügten Komponente, die durch neue Kanäle verbunden werden sollen.

Nach dem Abschluss des dritten Teilschritts wird eine Kopie der ausgewählten Komponente in das Kanalbündel eingefügt und ihre Ports werden durch neue Kanäle mit den Ports des Kanalbündels verbunden. Eine ausführliche Beschreibung dieses Vorgangs findet sich in [Tracht] (S.40-47).

Man kann sich leicht eine Situation vorstellen, in der der Benutzer im dritten Teilschritt ein Dutzend Portzuordnungen getroffen hat und sich dann überlegt, dass er im ersten Schritt noch einen Kanal dazunehmen will. In diesem Fall kann der Benutzer mithilfe des *Previous*-Buttons im Eingabedialog zum zweiten und dann zum ersten Teilschritt zurückkehren, die Kanalauswahl anpassen und zum dritten Teilschritt weitergehen: die Eingaben, die er im dritten Eingabedialog vorgenommen hat, gehen dabei nicht verloren.

Dieser Vorgang lässt sich zwar zurzeit nicht mithilfe einer ODL-Abfrage programmieren, dies sollte aber voraussichtlich mit den nächsten Erweiterungen der ODL-Ausdrucksmächtigkeit möglich werden. Nehmen wir also an, dass es eine ODL-Abfrage gibt, die diesen Vorgang beschreibt (eine

Abfrage, die keine Kopie, sondern die Original-Komponente in ein Kanalbündel einsetzt, ist bereits mit dem aktuellen Sprachumfang von ODL möglich). Die aktuelle Implementierung der Dialogflusskontrolle im ODL-Query-Subsystem erlaubt die Rückkehr zu früheren Eingabedialogen, speichert allerdings die Eingaben aus späteren Eingabedialogen nicht. Wenn der Benutzer also vom dritten Teilschritt zum ersten zurückkehrt, gehen die im dritten Teilschritt getroffenen Portzuorndnungen verloren, sodass er alle Eingaben wiederholen müsste, was nicht besonders benutzerfreundlich ist.

Um hier Abhilfe zu schaffen, muss die Dialogflusskontrolle des ODL-Query-Subsystems erweitert werden. Dafür gibt es zwei Möglichkeit, die sich in Implementierungsaufwand und der resultierenden Flexibilität unterscheiden. Wir wollen sie im folgenden besprechen und ihre Vorteile und Nachteile erörtern.

1) Die einfachere Möglichkeit ist eine Erweiterung des bestehenden Systems, indem jeder `context`-Quantor sich den zuletzt eingegebenen Wert merkt und, falls ein Rückwärtsschritt vorgenommen wurde, bei der Rückkehr zu ihm diesen Wert anzeigt. Dabei muss allerdings eine Analyse vorgenommen werden, ob der gespeicherte Eingabewert in der jetzigen Situation überhaupt möglich wäre. Betrachten wir die Beispielabfrage

```
context c:Component. context p:element( c.Ports ). true
```

Falls bei dieser Abfrage der Benutzer vom zweiten `context`-Quantor zum ersten zurückkehrt, dort eine andere Komponente auswählt und dann wieder zum zweiten `context`-Quantor übergeht, so wird der früher ausgewählte Port nicht mehr gültig sein, weil er zur früher ausgewählten Komponente, nicht aber zur aktuell ausgewählten Komponente gehört und deshalb jetzt gar nicht ausgewählt werden könnte. In einer solchen Situation muss der Eingabedialog für den zweiten `context`-Quantor mit leerer Auswahl starten.

Solche Gültigkeitsprüfungen müssen zwar bei jedem Wiedereintritt in einen zuvor verlassenen Eingabedialog vorgenommen werden, stellen aber keine große Schwierigkeit dar, denn es ist bekannt, zu welchem Typ ein einzugebender Wert gehört (es ist gerade der Typ der abgefragten Variablen), und es ist ganz einfach möglich, für einen abgespeicherten Wert zu überprüfen, ob er zu einem gegebenen Typ gehört: je nach Datentyp muss man über alle Typwerte iterieren und sie mit dem abgespeicherten Wert vergleichen oder es muss, falls es sich um einen ODL-Grundtyp handelt (z.B. Boolean oder Integer), gar keine Überprüfung vorgenommen werden.

2) Eine wesentlich aufwändiger zu implementierende und gleichzeitig viel flexiblere Lösung würde das Konzept von Kontrollklassen zur Auswertung von ODL-Abfragen darstellen. Im Rahmen des bereits angesprochenen Systementwicklungsprojekts wurde eine auf den konkreten Anwendungsfall spezialisierte Kontrollklasse entwickelt, die für den dreischrittigen Vorgang der Kanalbündelauftrennung die Navigation zwischen den Eingabedialogen für die einzelnen Teilschritte erlaubte, bei der sowohl Rückwärtsschritte mit Beibehaltung von getätigten Eingaben als auch Vorwärtsschritte möglich waren ([Tracht], S.40-47). Bei einem Vorwärtsschritt wurden die abgespeicherten Eingaben hinsichtlich ihrer Kompatibilität mit den neuen Eingaben aus vorherigen Teilschritten überprüft und ungültig gewordene Teile der abgespeicherten Eingaben gelöscht. Für ODL-Abfragen gestaltet sich die Situation komplizierter, weil hier beliebige Abläufe von Benutzereingaben möglich sein sollen – eine ähnlich flexible Navigation ist dementsprechend schwieriger zu implementieren.

In der aktuellen Version wird der Durchlauf durch einen ODL-Auswertungsbaum über Aufrufe von Methoden implementiert: der Auswertungsbaum ist, in gewissem Sinne, im Java-Callstack versteckt und für den expliziten Zugriff nicht ohne Weiteres zugänglich: die einzige Zugriffsmöglichkeit stellen Exceptions zum Abbruch einer momentan ausgeführten Methode dar – dieser Mechanismus wird in der aktuellen Implementierung für die Rückkehr zu vorherigen Eingabedialogen benutzt.

Um die Steuerung des Dialogflusses durch Kontrollklassen zu ermöglichen, müsste die Steuerung

des Durchlaufs durch einen ODL-Auswertungsbaum auf die Kontrollklasse übertragen werden – zumindest für die Auswertung von `context`-Quantoren. Der jetztige Auswertungsmechanismus, der komplett über den Java-Callstack läuft und keine Eingriffspunkte während der Auswertung biete, müsste insofern aufgebrochen werden, dass die `evaluate`-Methode der `ContextQuantifier`-Klasse die Kontrolle nicht direkt an den Eingabedialog, sondern an die Kontrollklasse übergibt. Die Kontrollklasse führt die Eingabe aus, indem sie den jeweiligen Eingabedialog startet, und speichert den eingegebenen Wert in einer lokalen Hashtabelle zusammen mit einer Referenz auf den aufrufenden `context`-Quantor ab. Anschließend übergibt sie den eingegebenen Wert und die Ausführungskontrolle zurück an den `context`-Quantor.

Mit diesem Konzept wird die Benutzereingabe und Dialogflusskontrolle von der `ContextQuantifier`-Auswertungsklasse abgekoppelt. Auf diese Art und Weise könnte eine Kontrollklasse die Auswertung des ODL-Auswertungsbaums prizipiell an einem beliebigen `context`-Quantor unterbrechen und an einem beliebigen anderen starten – die Grundlage dafür ist die Verwaltung aller Zustandsinformationen durch die Kontrollklasse selbst.

Die Implementierung eine Dialogflusskontrolle, mit welcher der Benutzer Rückwärtsschritte und anschließende Vorwärtsschritte ausführen kann, bei denen die getätigten Eingaben nicht verloren gehen, wäre nur eine der Möglichkeiten, die Kontrollklassen bieten. Denkbar wären auch folgende Implementierungen der Dialogflusskontrolle:

- Der Benutzer kann aus einer Liste vorheriger Schritte direkt auswählen, zu welchem Schritt er zurückkehren will.
- Für ein und denselben Eingabeschritt werden mehrere Zustände gespeichert, sodass nicht nur die zuletzt getätigte Eingabe, sondern auch eine der früher gemachten Eingaben wieder hergestellt werden kann: der Benutzer könnte aus einer Liste abgespeicherter Eingaben die gewünschte auswählen.
- Test-Kontrollklassen, bei denen Benutzereingaben nicht vom Benutzer vorgenommen, sondern von einem Testfall-Generator erzeugt werden. Die Test-Kontrollklasse kann sogar insofern vom Benutzer kofigurierbar sein, dass er vor Beginn des Tests eine Regel für die Erzeugung von Testfällen vorgibt.

In allen beschriebenen Fällen müssten andere ODL-Auswertungsklassen nicht modifiziert werden, weil alle Kontrollfunktionalitäten in der verwendeten Kontrollklasse gekapselt sind, die nach außen eine defininierte Schnittstelle aufweist.

Eine weitere Möglichkeit würde durch die Anwendung des *Strategie*-Entwurfsmusters auf Kontrollklassen eröffnet, wie es bereits zur Konfiguration von Eingabedialogen benutzt wird. Dann könnte der Benutzer zur Laufzeit über einen Konfigurationsdialog festlegen, welche der zur Auswahl stehenden Kontrollklassen verwendet werden soll – beispielsweise könnte zwischen einer Standard-Kontrollklasse (Benutzereingabe mit Möglichkeit von Rückwärtsschritten) und einer Test-Kontrollklasse (keine Eingabe durch den Benutzer, sondern Erzeugung von Testwerten durch einen Testfall-Generator) gewählt werden.

Die erste beschriebene Lösung lässt sich mit mittlerem Aufwand implementieren, bietet jedoch viele der Möglichkeit der zweiten Lösung nicht, insbesondere ist sie nicht beliebig erweiterbar und kann zur Laufzeit nur mit Schwierigkeiten konfiguriert werden.

Die zweite Lösung erfordert einen höheren Implementierungsaufwand, ist aber wesentlich flexibler und bietet zahlreiche Weiterentwicklungsmöglichkeiten, darunter die Konfiguration der Dialogflusskontrolle zur Laufzeit und das Hinzufügen neuer Kontrollklassen, ohne dass andere ODL-Auswertungsklassen modifiziert werden müssen.

Kapitel 7

Fazit

Die vorliegende Arbeit setzte sich zum Ziel, eine Erweiterung der *Operation Definition Language (ODL)* im AutoFocus/Quest-Application-Framework vorzunehmen und eine interaktive Benutzerschnittstelle für die Auswertung von ODL-Abfragen zu implementieren. Diese Ziele wurde im Rahmen der in der in der Aufgabenstellung (Abschnitt 1.1) beschriebenen Anforderungen erreicht:

- Der Sprachumfang von ODL wurde um die beschriebenen Konstrukte – Produkttypen, Mengenkomprehension, Mengentypen, Selektorausdrücke, benamte Prädikate – sowie um einige zusätzliche Konstrukte, darunter Vergleiche und arithmetische Operationen, erweitert.

- Für Benutzereingaben wurde eine interaktive dialogbasierte Benutzerschnittstelle entwickelt, welche die Eingabe von Werten für alle zurzeit unterstützten ODL-Datentypen ermöglicht und während eines Eingabevorgangs die Rückkehr zu früheren Eingabeschritten erlaubt.

Bei der Implementierung neuer ODL-Sprachkonstrukte sowie der interaktiven Benutzerschnittstelle wurde auf die Wiederverwendbarkeit der erstellten Module sowie die Flexibilität und Erweiterbarkeit des ODL-Auswertungssystems und, als sein Bestandteil, der interaktiven Benutzerschnittstelle geachtet. Insbesondere bedürfen Modifikationen der Benutzerschnittstelle, die nicht mit einer Änderung der ODL-Sprachkonstrukte einhergehen, keiner Anpassungen anderer Bestandteile des ODL-Auswertungssystems.

Zusätzlich zu den realisierten Erweiterungen von ODL wurden im Kapitel 6 Vorschläge für Optimierungen und Erweiterungen von ODL gemacht, die aufgrund ihres Umfangs nicht im Rahmen dieser Arbeit implementiert werden konnten.

Wir wollen an dieser Stelle noch einige weiter reichende Ideen für die Weiterentwicklung des ODL-Systems anführen:

→ Bibliotheken von ODL-Abfragen:
Dem Benutzer können Sammlungen von ODL-Abfragen und Bausteinen für ODL-Abfragen für gebräuchliche Konsistenzprüfungen und Operationen in Form von Bibliotheken zur Verfügung gestellt werden. Es sind auch ODL-Templates denkbar, bei denen die Struktur der Abfrage vorgegeben ist und nur bestimmte Datentypen oder Operationen vom Benutzer ergänzt werden sollen.

→ Erweiterung der Auswahlmechanismen für Benutzereingaben:
In der aktuellen ODL-Version werden Benutzereingaben für Metamodell-Typen durchgeführt, indem eine Liste aller verfügbaren Entitäten des betreffenden Typs angezeigt wird, aus welcher der Benutzer einen Eintrag auswählen soll. Die Eingabe lässt sich für den Benutzer bequemer gestalten, wenn der Auswahlmechanismus die in QUEST verwendete Modellansicht (zurzeit eine Baumansicht) einbezieht – ein Modellelement kann dann direkt im Modelleditor ausgewählt werden.

In zukünftigen Entwicklungen des AutoFocus/Quest-Frameworks ist es denkbar, dass für die Bearbeitung des Modells im QUEST-Werkzeug ein graphischer Editor implementiert wird, wie er bereits in AutoFocus existiert. In diesem Fall können Auswahlmechanismen für ODL-Benutzereingaben auch auf den graphischen Editor erweitert werden – bei der Eingabe eines Modell-Elements könnte der Benutzer das Modell-Element direkt im graphischen Editor auswählen. Möglich ist auch die Kombination der aktuellen dialogbasierten Auswahlmechanischen mit graphischen Auswahlmechanismen, sodass die Auswahl vom Modellelementen parallel im Eingabedialog und im graphischen Editor ablaufen könnte.

Schließlich ist auch die Darstellung von Ergebnissen einer ODL-Abfrage in der QUEST-Modellansicht denkbar, indem Modellelemente, die im Abfrageergebnis enthalten sind, in der Modellansicht hervorgehoben werden.

→ Syntax-Highlighting für ODL-Abfragen:
Um die Erstellung von Abfragen im ODL-Editor zu erleichtern, kann eine Syntaxhervorhebung eingeführt werden, wie sie in den meisten Entwicklungumgebungen für Programmiersprachen üblich ist.

Abschließend kann gesagt werden, das ODL auf dem aktuellen Entwicklungsstand ein flexibles und vielseitiges Werkzeug zur Validierung und Transformation von konzeptmodellbasierten Modellen dartellt, das ein großes Weiterentwicklungspotenzial bereithält.

Anhang A

Klassendiagramme

In diesem Anhang werden Klassendiagramme des Pakets `quest.odl.evaluation` und ausgewählte Klassendiagramme des Pakets `quest.dialogs` aufgeführt. Folgende Klassendiagramme sind enthalten:

Abbildung A.1:	Package `quest.odl.evaluation`: Ausgangspackage für Klassen zur Auswertung von ODL-Abfragen
Abbildung A.2:	Package `quest.odl.evaluation.generator`: `SableCCGenerator` für ODL-Abfragen sowie von ihm benutzte Klassen
Abbildung A.3:	Package `quest.odl.evaluation.model`: `Expression`-Hierarchie
Abbildung A.4:	Package `quest.odl.evaluation.model`: `Term`-Hierarchie
Abbildung A.5:	Package `quest.odl.evaluation.model`: `MetaType`-Hierarchie
Abbildung A.6:	Package `quest.odl.evaluation.model`: `TermResult` und Exception-Klassen
Abbildung A.7:	Package `quest.util.collections`: Kollektionen und Iteratoren
Abbildung A.8:	Package `quest.odl.evaluation.model.analysis`: `EvalTreeVisitor`-Hierarchie und das Interface `EvalTreeNode`
Abbildung A.9:	Package `quest.odl.evaluation.model.query`: Query-Klassen
Abbildung A.10:	Package `quest.odl.evaluation.model.query.dialog`: `QueryInputPanel`-Hierarchie
Abbildung A.11:	Package `quest.odl.evaluation.model.query.dialog`: `QueryDialog`-Hierarchie
Abbildung A.12:	Package `quest.odl.evaluation.model.cellRenderers.formatters`: Cellrenderer-Klassen und `ObjectToStringFormatter`-Hierarchie
Abbildung A.13:	Package `quest.dialogs.navigationBar`: `NavigationBar` und benutzte Klassen
Abbildung A.14:	Package `quest.odl.evaluation.model.query.factory`: `QueryInputPanelFactory`-Hierarchie
Abbildung A.15:	Package `quest.odl.evaluation.model.query.factory`: `QueryInputPanelProducer`-Hierarchie
Abbildung A.16:	Package `quest.odl.evaluation.model.query.factory`: `InputVerifier`-Klassen, `ObjectToStringFormatterFactory`-Hierarchie und weitere Klassen

Evaluation

-Evaluation:
+evaluate:EvaluationResult

EvaluationModel

-term:Term

EvaluationModel:
+evaluate:EvaluationResult
+toString:String

interface
EvaluationResult

+existsSatAssignment:boolean
+getSatAssignments:Iterator

EvaluationModelGenerator

-generator:Generator

-EvaluationModelGenerator:
+createEvaluationModel:EvaluationModel

BoundVariable

+identifier:String
+value:Object

+BoundVariable:
+toString:String

EvaluationException

+EvaluationException:
+EvaluationException:

UserBreakException

+MESSAGE:String

+UserBreakException:
+UserBreakException:

quest.odl.evaluation.generator

+InvalidOperandTypeException
+InvalidSelectorUsageException
+InvalidExpressionUsageException
+UndefinedNamedPredicateException
+InvalidArgumentsException
+UndefinedSelectorException
+UnsupportedConstructException
+UndefinedVariableException
+UndefinedTypeException
+UndefinedRelationException
+SableCCGenerator
+QUESTModelMapper
+ParserException
+LexerException
+IncompatibleTypesException
+GeneratorException
+Generator
+AmbiguityException

quest.odl.evaluation.model

+MultiplicationExpression
+SubtractionExpression
+BinaryArithmeticExpression
+AdditionExpression
+SmallerExpression
+BiggerExpression
+ComparisonExpression
+SetSizeExpression
+ProductValue
+CompositeValue
+EmptySetTest
+AssociatedEntitiesSelector
+AssociatedEntitySelector
+AttributeSelector
+CompositeTypeSelector
+MetaRestrictedType
+NamedPredicateTerm
+InvalidArgumentsListException
+NamedPredicate
+RelationTerm
+Quantifier
+Junction
+MetaIntroducedType
+SetValue
+AbstractMetaCompositeType
+MetaSetType
+AbstractMetaType
+SelectorExpression
+NoSuchSelectorException
+DuplicateIdentifierException
+MetaCompositeType
+InvalidProductTypeException
+MetaProductType
+Selector
+UniversalQuantifier
UniversalExistentialQuantifier
+Variable
+UnboundVariableException
+TermResult
+Term
+SimpleLinkedList
+RelationTestTerm
+RelationRemovalTerm
+RelationRemoval
+RelationModification
+RelationAdditionTerm
+RelationAddition
+Relation
+NoSuchRelationException
+NoSuchConstructorException
+UndefinedModelElementException
+NewQuantifier
+Negation
+MetaType
+MetaString
+MetaInteger
+MetaEntity
+ReflectionException
+InstanceRetrievalException
+Implication
+Expression
+ExistentialQuantifier
+EvaluationModelException
+Equivalence
+EqualExpression
+Entity
+EmptyIterator
+Disjunction
+ContextQuantifier
+Constant
+Conjunction
+BoolConstantTerm
+Binding
+AttributeRelation
+MetaBool
+Assignment
+AssociationRelation

Abbildung A.1: Package `quest.odl.evaluation`

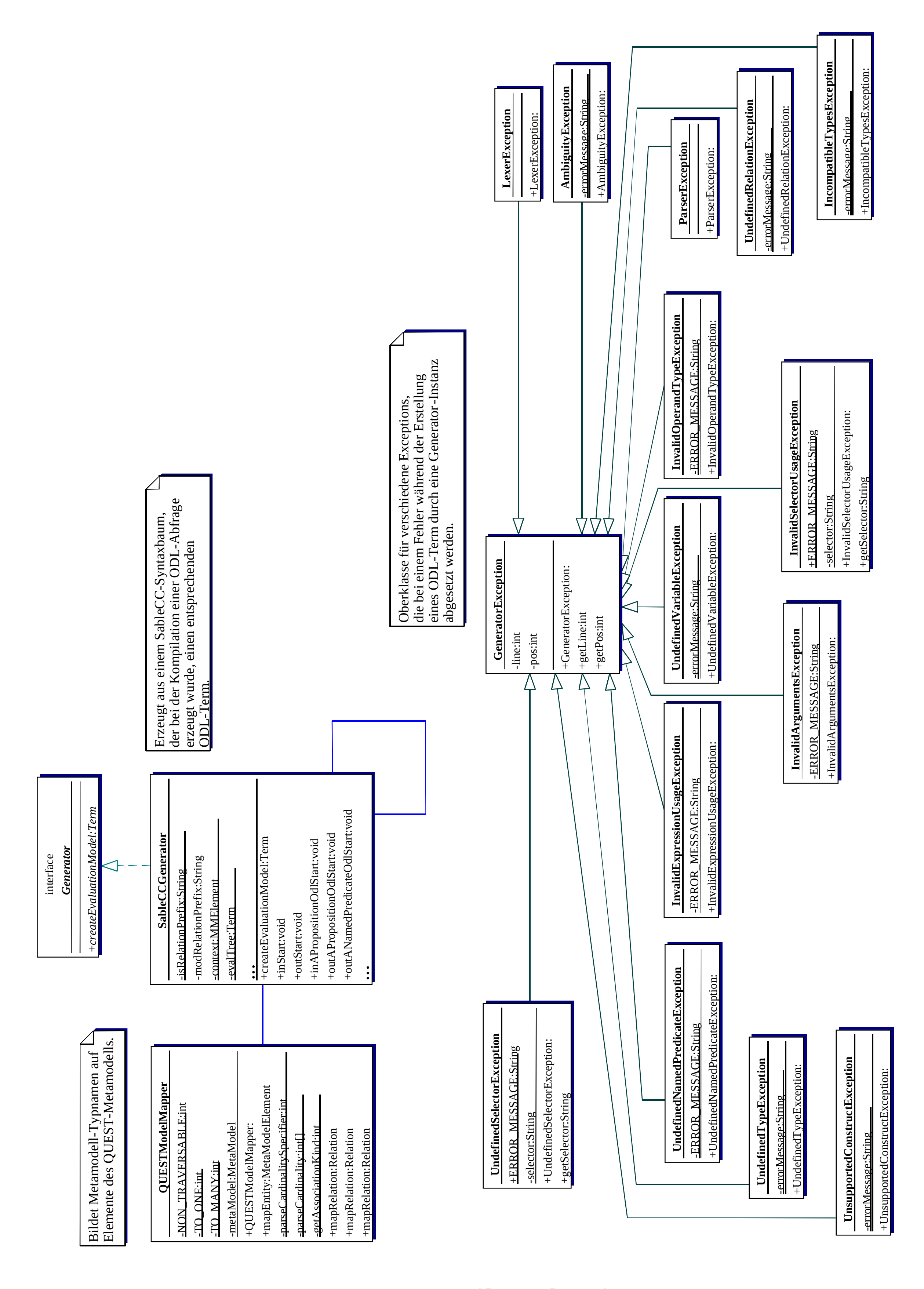


Abbildung A.2: Package `quest.odl.evaluation.generator`

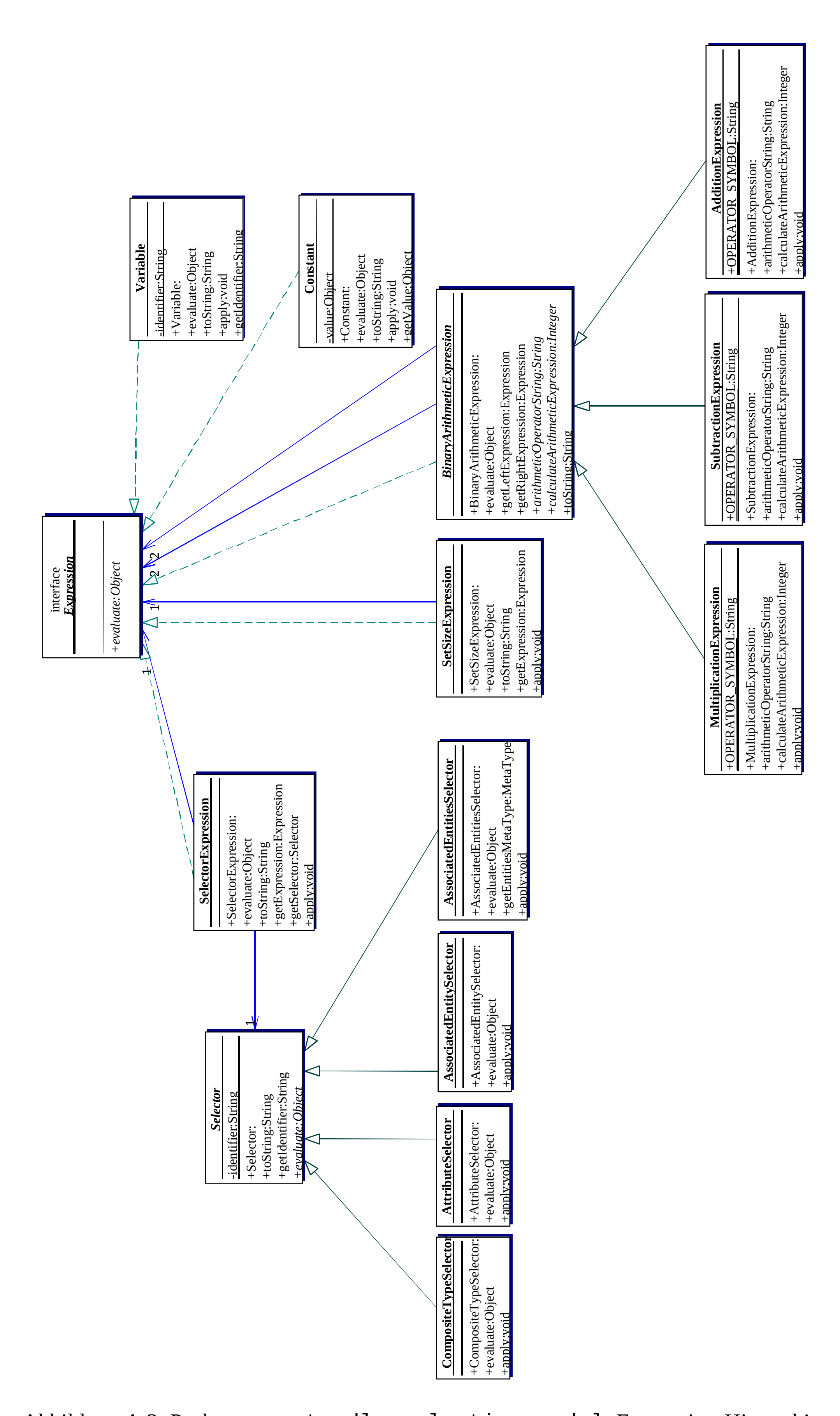


Abbildung A.3: Package `quest.odl.evaluation.model`: Expression-Hierarchie

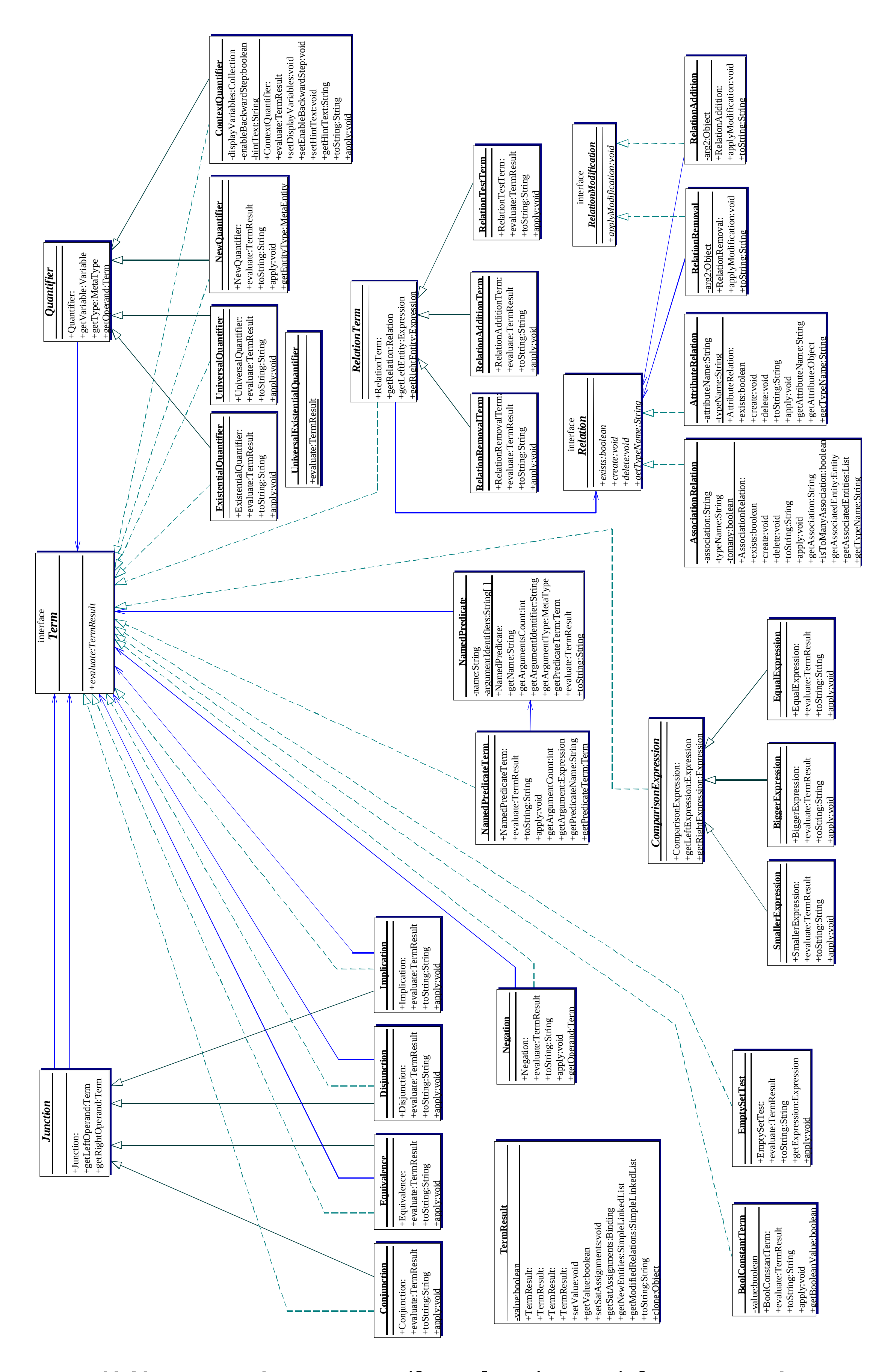


Abbildung A.4: Package `quest.odl.evaluation.model`: Term-Hierarchie

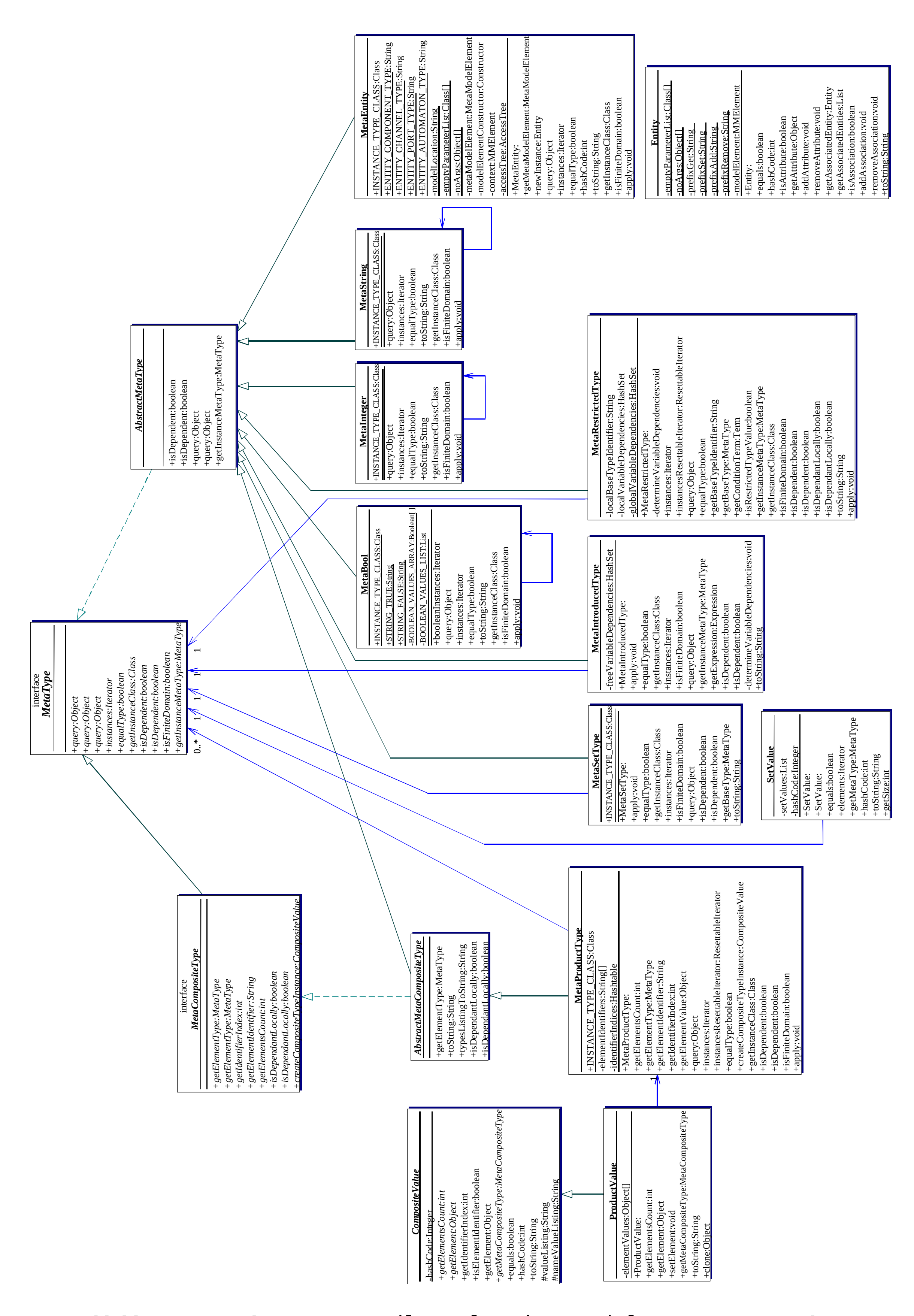


Abbildung A.5: Package `quest.odl.evaluation.model`: MetaType-Hierarchie

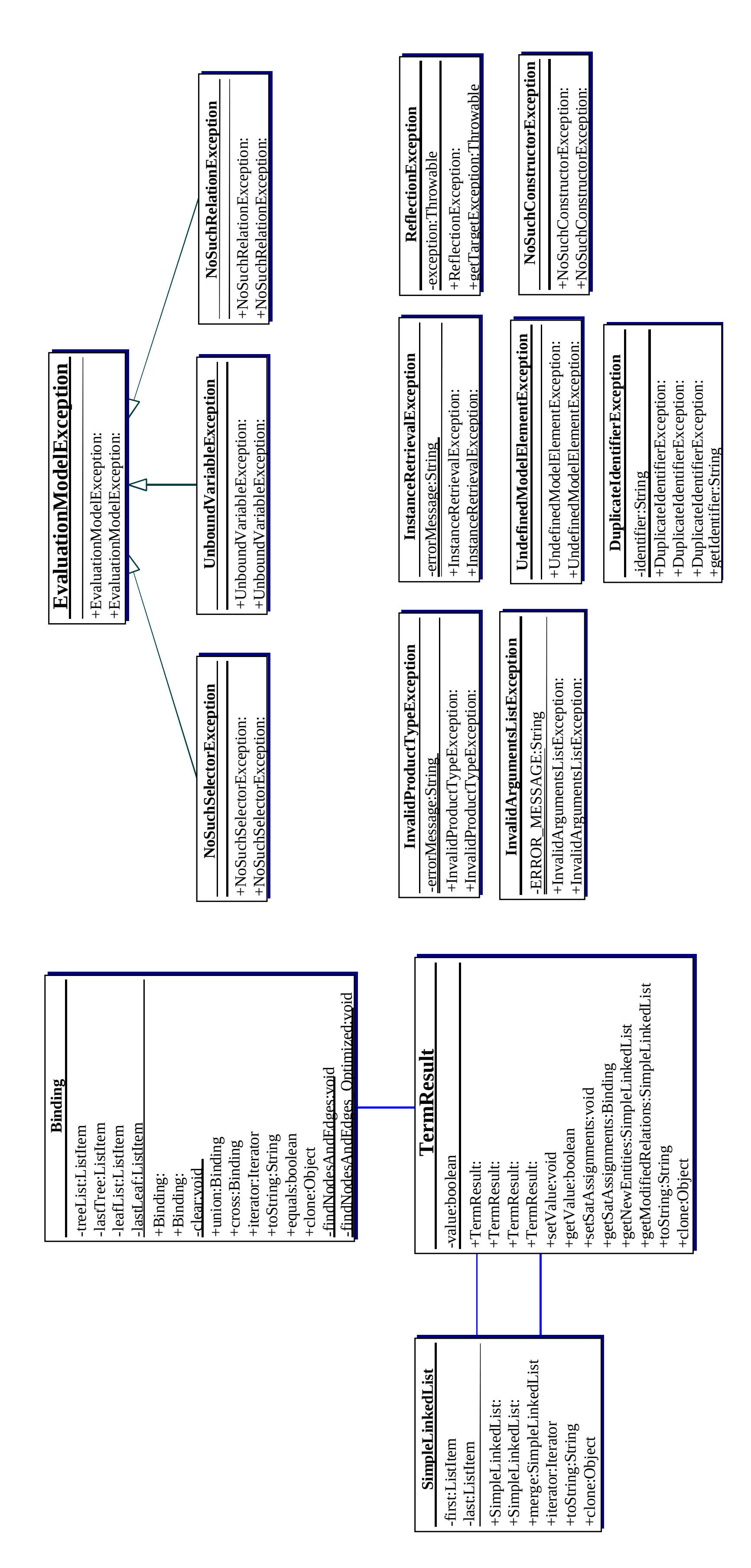


Abbildung A.6: Package `quest.odl.evaluation.model`: TermResult und Exceptions

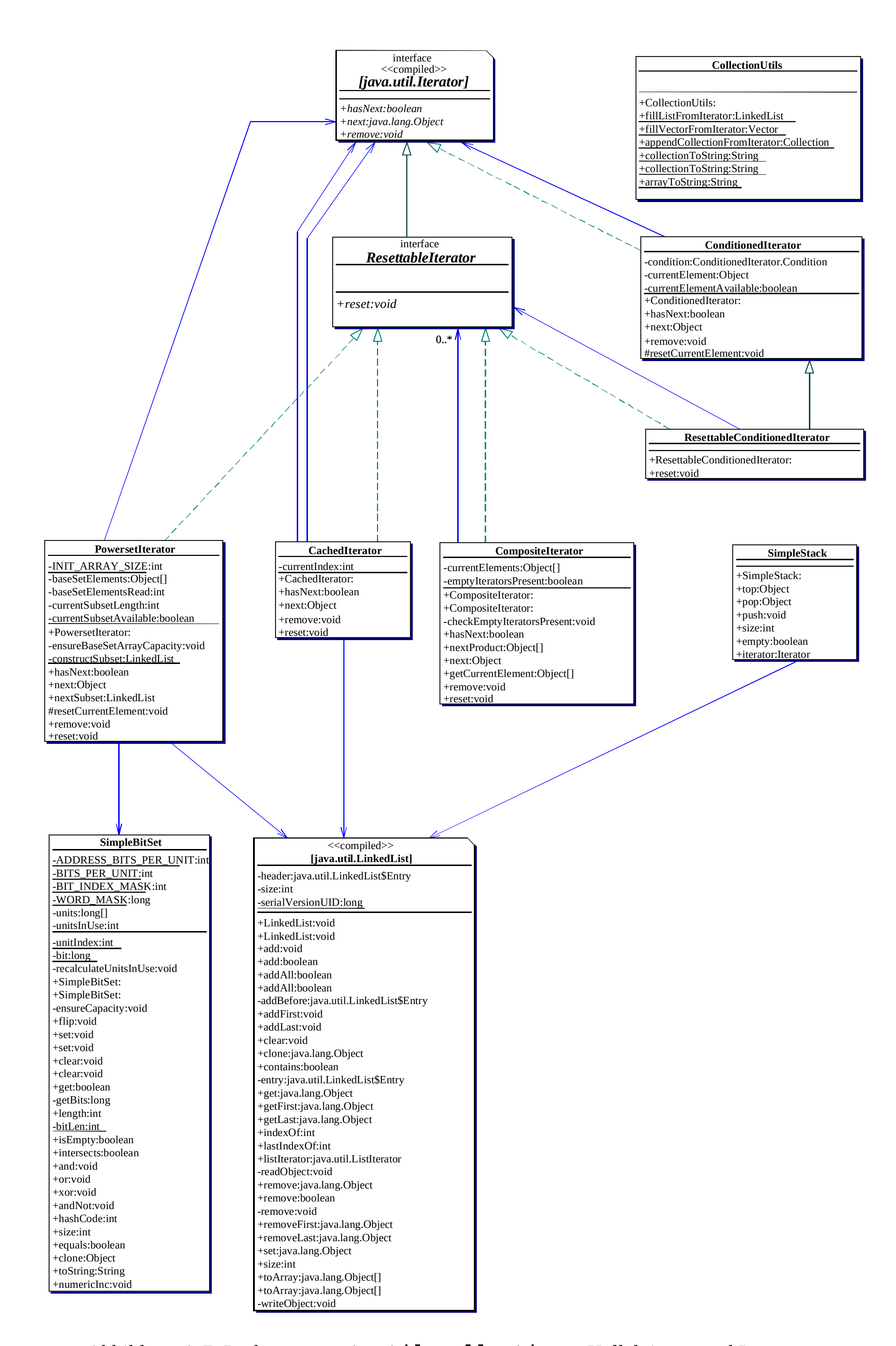


Abbildung A.7: Package `quest.util.collections`: Kollektionen und Iteratoren

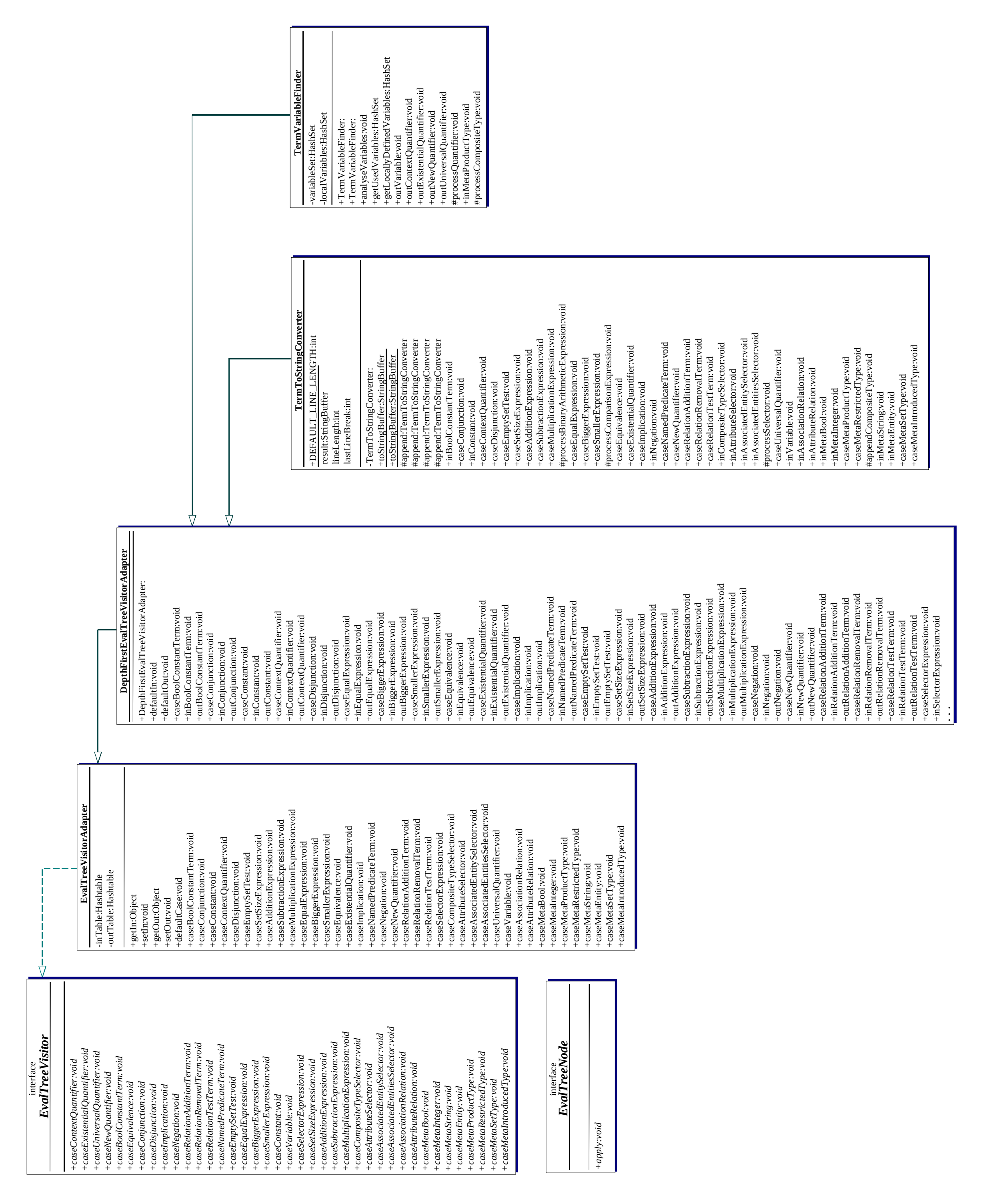


Abbildung A.8: Package `quest.odl.evaluation.model.analysis`: EvalTreeVisitor-Hierarchie und EvalTreeNode

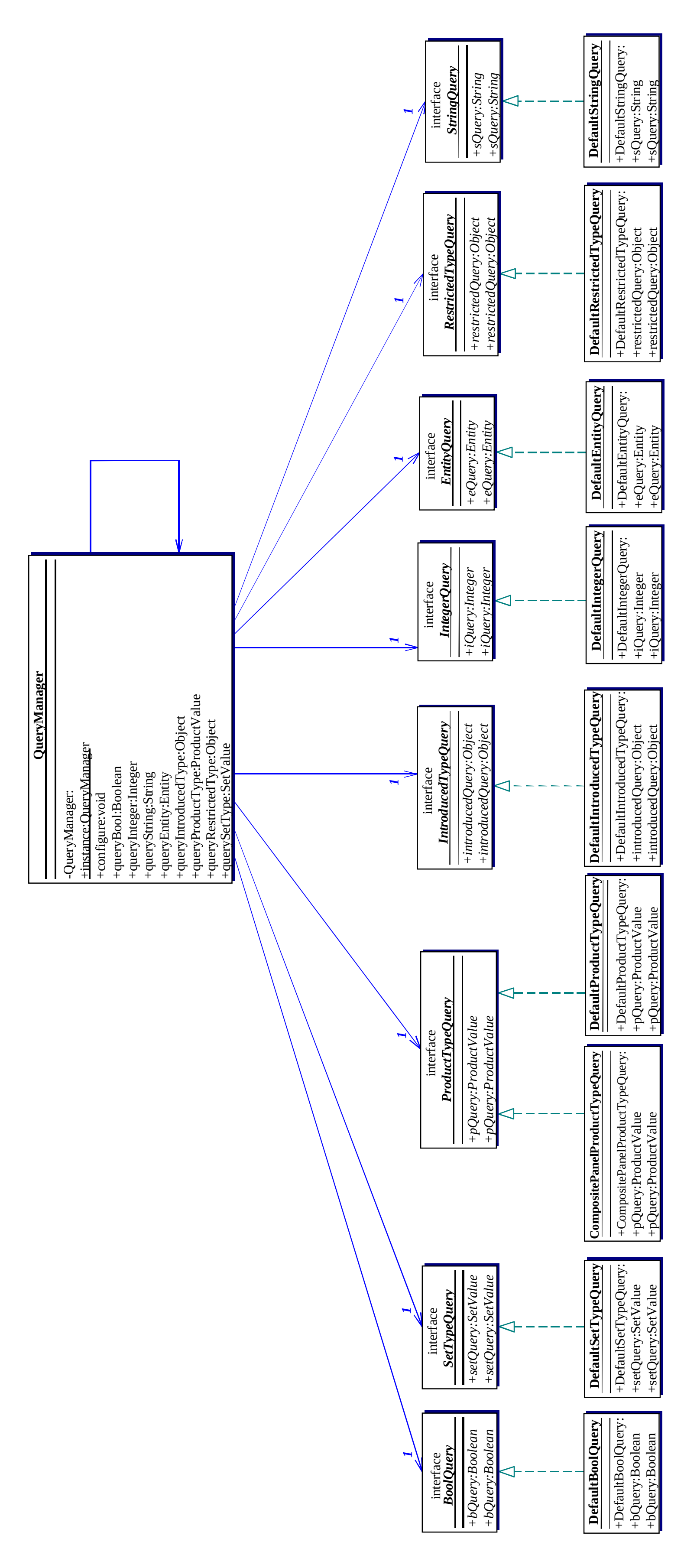


Abbildung A.9: Package `quest.odl.evaluation.model.query`

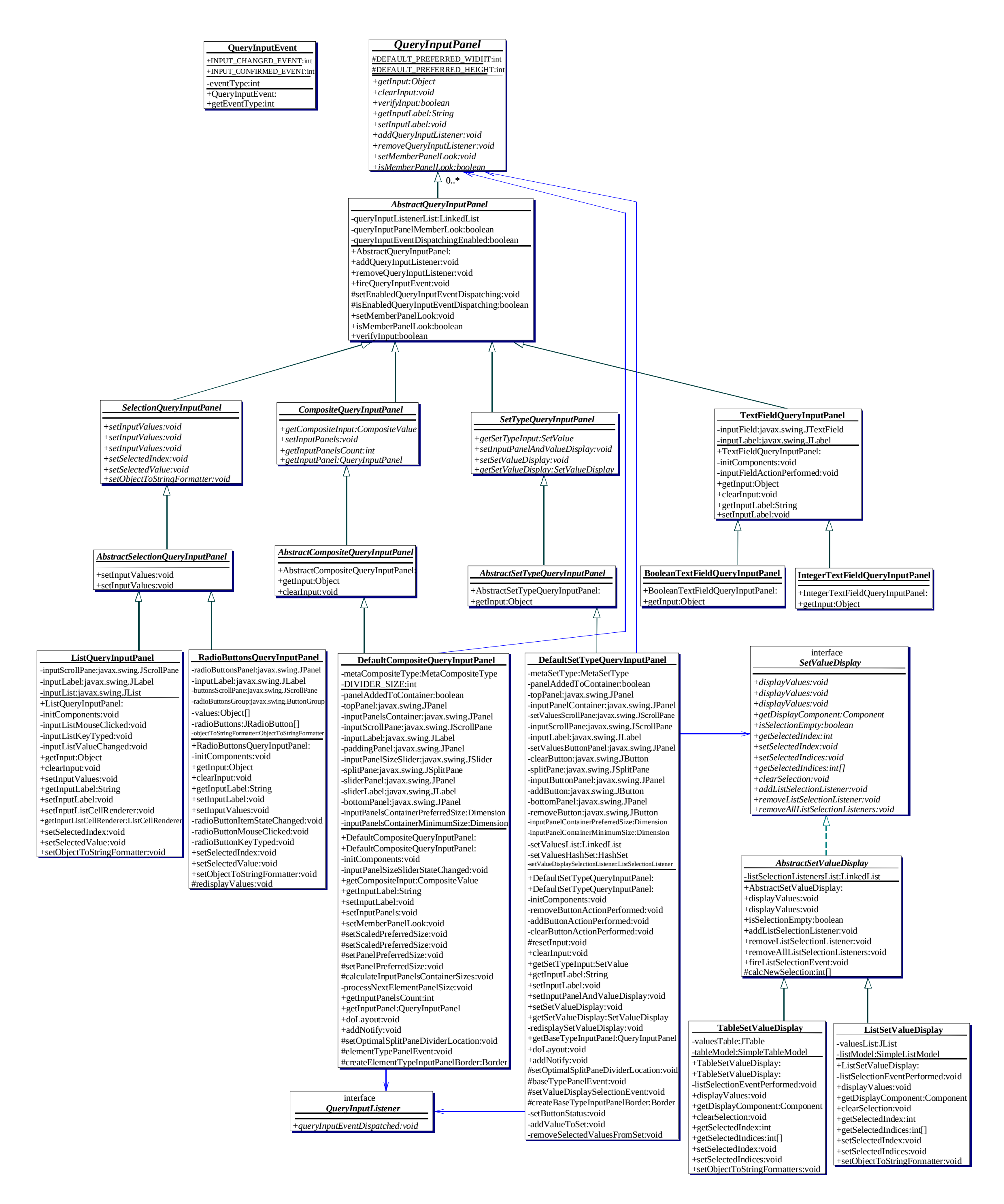


Abbildung A.10: Package `quest.odl.evaluation.model.query.dialog`: QueryInputPanel-Hierarchie

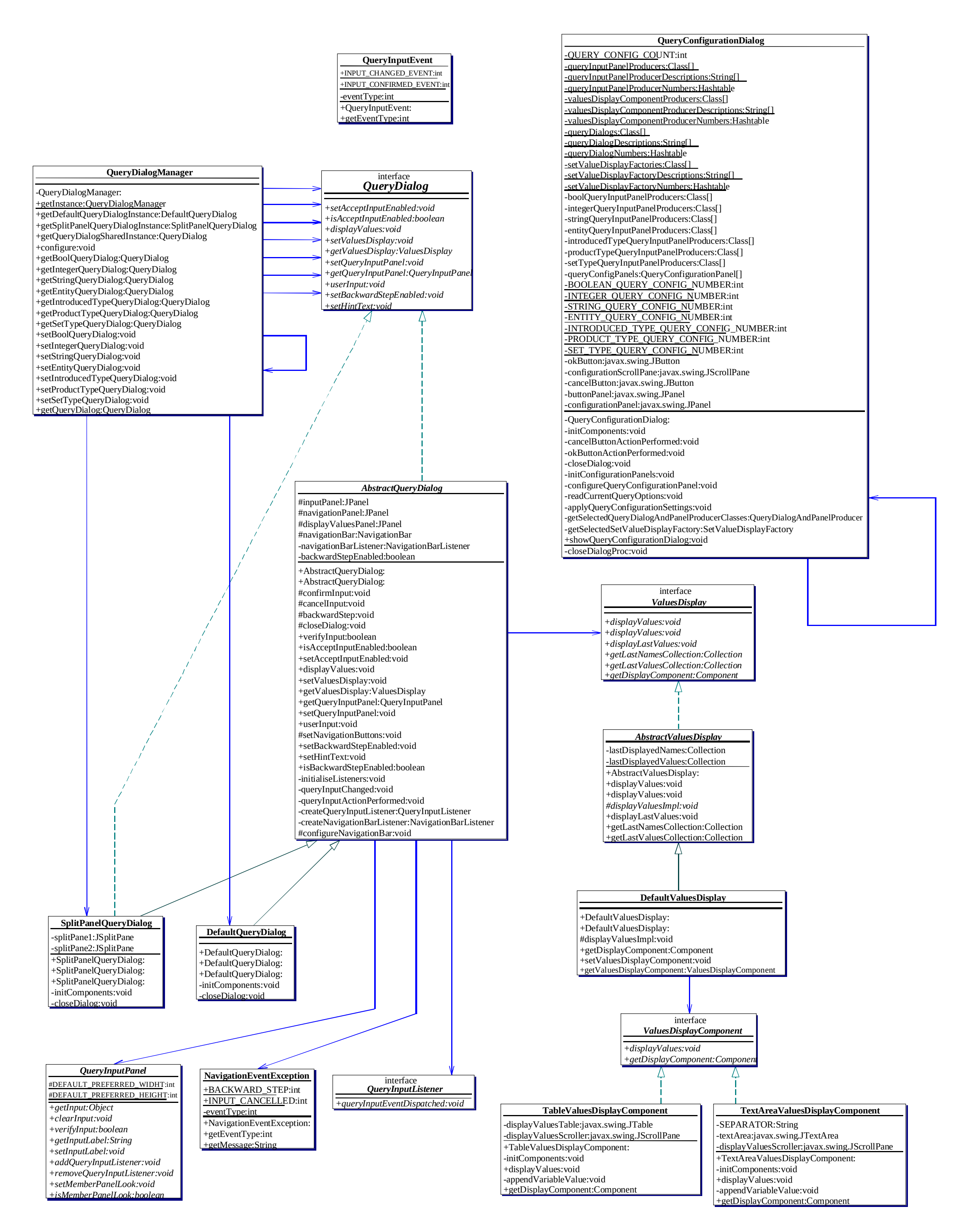


Abbildung A.11: Package `quest.odl.evaluation.model.query.dialog`: QueryDialog-Hierarchie

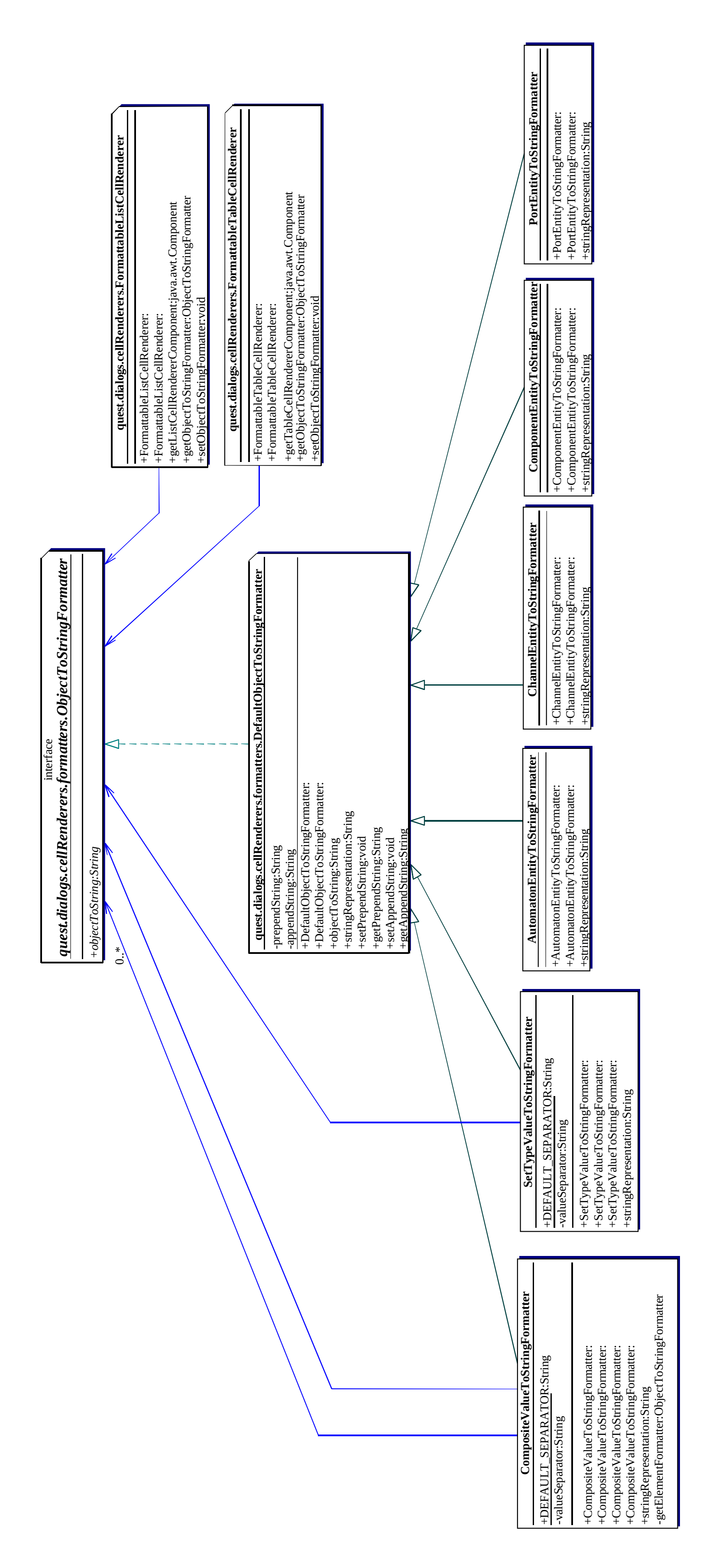


Abbildung A.12: Package `quest.odl.evaluation.model.cellRenderers.formatters`: Cellrenderer-Klassen und ObjectToStringFormatter-Hierarchie



Abbildung A.13: Package `quest.dialogs.navigationBar`: `NavigationBar` und benutzte Klassen

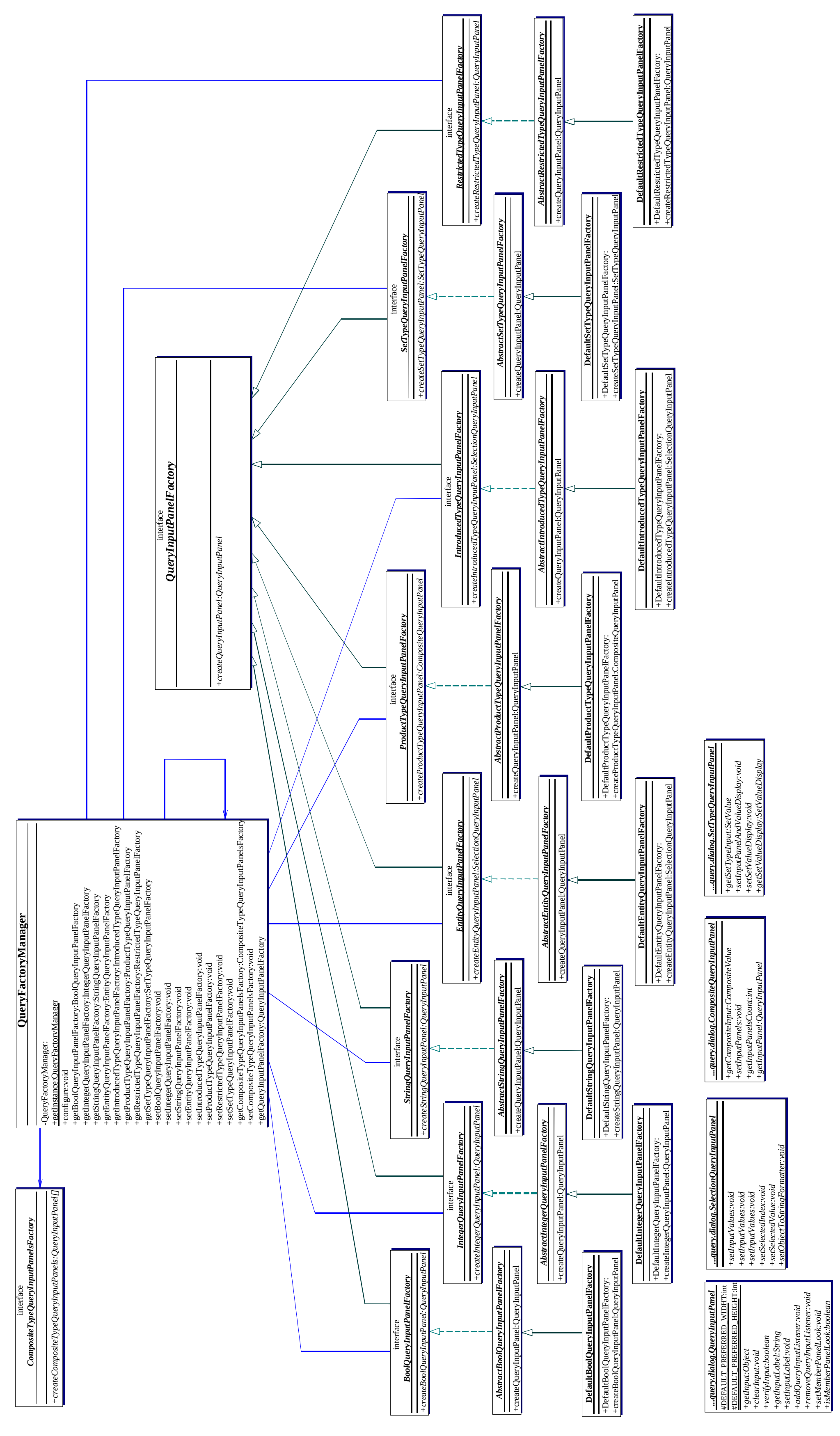


Abbildung A.14: Package `quest.odl.evaluation.model.query.factory`: QueryInputPanelFactory-Hierarchie

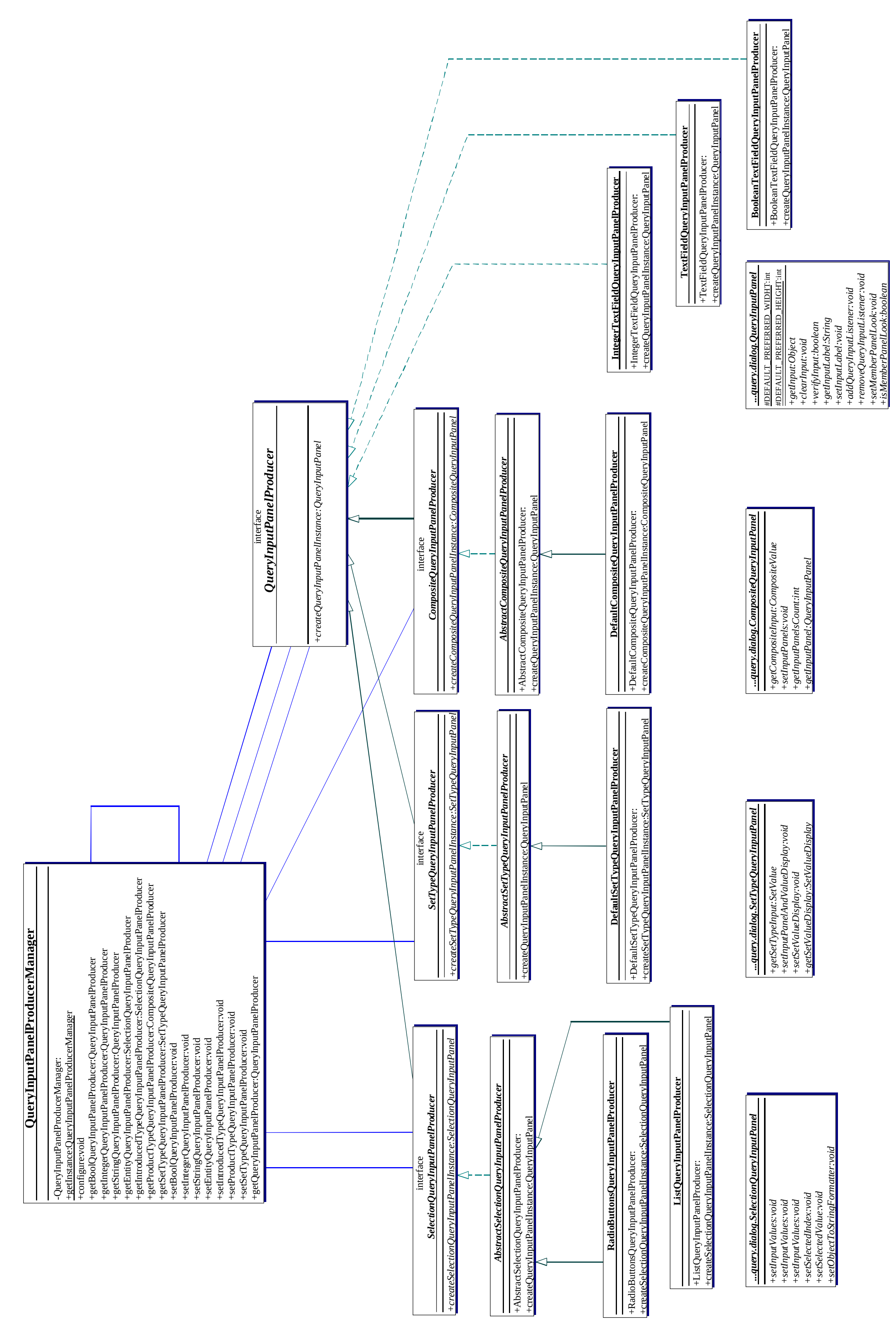


Abbildung A.15: Package `quest.odl.evaluation.model.query.factory`: QueryInputPanelProducer-Hierarchie

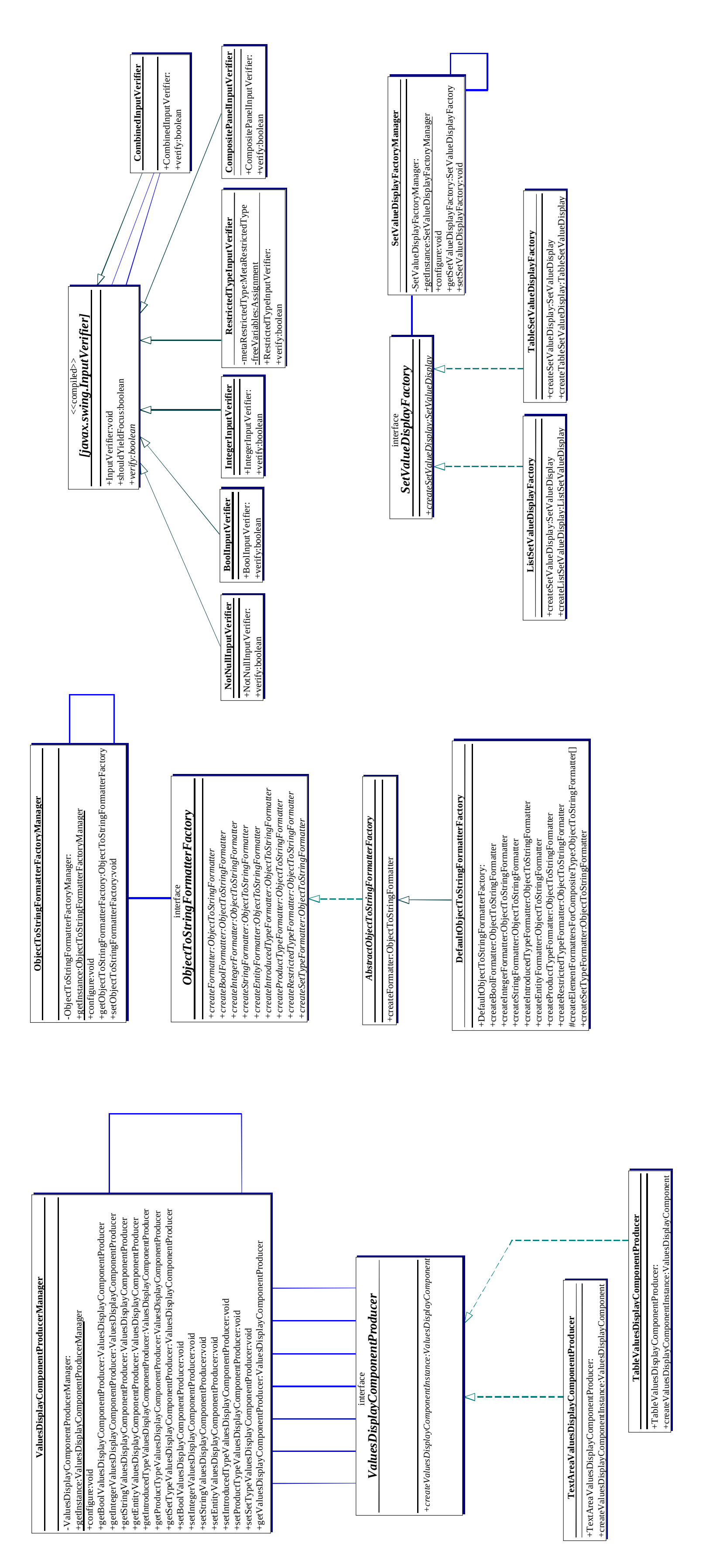


Abbildung A.16: Package `quest.odl.evaluation.model.query.factory`: InputVerifier, ObjectToStringFormatterFactory und weitere Klassen

Anhang B

ODL-Grammatik

In diesem Anhang befindet sich die formale ODL-Grammatik. Sie stellt eine Erweiterung der in [Sch01] (S.16-19) vorgestellten ODL-Grammatik dar.

Bemerkungen:

- Terminale Symbole sind **fett** gedruckt
- Terminale Symbole, die von der lexikalischen Analyse zusammengefasst werden, sind *kursiv* gedruckt (z.B. *ident*, *bool_constant*)
- Alle anderen Symbole sind nicht-terminal.
- Diese Grammatik wurde mit Hinblick auf die Lesbarkeit definiert. Für eine technische Implementierung könnten andere Produktionsregeln von Vorteil sein.

start	::=	proposition |
	::=	named_predicate_declaration
named_predicate_declaration	::=	*ident* := CCL_proposition
proposition	::=	proposition **and** proposition |
		proposition **or** proposition |
		proposition **implies** proposition |
		proposition **equiv** proposition |
		neg proposition |
		(proposition **)** |
		basic_proposition |
		quantor_proposition |
		named_predicate_call
basic_proposition	::=	relation |
		comparison_expression |
		set_is_empty |
		bool_constant
relation	::=	pre_relation |
		post_relation

pre_relation	::=	relation_ident **(** arguments **)**
relation_ident	::=	*ident*
post_relation	::=	**result** relation_ident **(** arguments **)** | **result not** relation_ident **(** arguments **)**
set_is_empty	::=	**isEmpty(** expression **)**
arguments	::=	argumentlist | <empty>
argumentlist	::=	argument , argumentlist | argument
argument	::=	expression
expression	::=	functional_expression | selector_expression | arithmetic_expression | primitive_expression
functional_expression	::=	function_ident**(** arguments **)**
function_ident	::=	*ident*
selector_expression	::=	variable **.** selection
selection	::=	selector **.** selection | selector
selector	::=	*ident*
primitive_expression	::=	constant | variable
constant	::=	*bool_constant* | *int_constant* | *string_constant*
variable	::=	*ident*
comparison_expression	::=	expression comparison_operator expression
comparison_operator	::=	$=$ | $<$ | $\leq$ | $>$ | $\geq$
arithmetic_expression	::=	factor |

		arithmetic_expression $+$ factor |
		arithmetic_expression $-$ factor
factor	::=	arithmetic_term |
		factor $*$ arithmetic_term
arithmetic_term	::=	*int_constant* |
		(arithmetic_expression **)** |
		size(expression **)**
quantor_proposition	::=	**forall** variable_definition **.** proposition |
		exists variable_definition **.** proposition |
		context variable_definition **.** proposition |
		new variable_definition **.** proposition
named_predicate_call	::=	named_predicate_ident**(** arguments **)**
named_predicate_ident	::=	*ident*
variable_definition	::=	variable **:** type
type	::=	product_type |
		restricted_type |
		set_type |
		element introduced_type
product_type	::=	basic_type |
		(type_list **)**
type_list	::=	ident : type |
		type_list , ident : type
basic_type	::=	*type_ident*
restricted_type	::=	**{** ident : type | CCL_proposition **}**
set_type	::=	**set** type
introduced_type	::=	expression

Eine CCL-Proposition entspricht einer ODL-Proposition mit der Einschränkung, dass die Quantoren **context** und **new** sowie die Schlüsselwörter **result** und **result not** nicht verwendet werden dürfen.

Anhang C

ODL-Grammatik in der SableCC-Notation

In diesem Anhang befindet sich die SableCC-Grammatik für den ODL-Parser, der durch das SableCC-Tool aus der Grammatik automatisch erzeugt und von der `SableCCGenerator`-Klasse bei der Übersetzung von ODL-Abfragen verwendet wird.

```
Package quest.odl.parser;

/**
 * Grammar of CCL-Extension ODL
 *
 * 0.1 2001/09/28
 * @author David Pasch
 * 0.5 2003-08-09
 * @author David Trachtenherz
 */

Helpers
  all    = [0 .. 0xFFFF];
  letter = [['A' .. 'Z'] + ['a' .. 'z']];
  digit  = ['0' .. '9'];
  al     = [letter + '_'];
  alnum  = [digit + al];

  tab = 9;
  cr  = 13;
  lf  = 10;
  eol = cr lf | cr | lf;

  not_cr_lf = [all - [cr + lf]];
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];

  onelined_comment = '//' not_cr_lf*;
  multiline_comment = '/*' not_star* '*'+
                      (not_star_slash not_star* '*'+)* '/';
```

```
Tokens
  // Special Characters
  equal     = ’=’;
  assign    = ’:=’;
  l_par     = ’(’;
  r_par     = ’)’;
  l_brace   = ’’;
  r_brace   = ’’;
  l_bracket = ’[’;
  r_bracket = ’]’;
  v_line    = ’|’;
  dot       = ’.’;
  colon     = ’:’;
  comma     = ’,’;
  smaller   = ’<’;
  bigger    = ’>’;
  smaller_or_equal = ’<=’;
  bigger_or_equal  = ’>=’;
  plus      = ’+’;
  minus     = ’-’;
  mult      = ’*’;

  // Operators
  and     = ’and’;
  or      = ’or’;
  implies = ’implies’;
  equiv   = ’equiv’;
  neg     = ’neg’;
  forall  = ’forall’;
  exists  = ’exists’;
  context = ’context’;
  new     = ’new’;
  result  = ’result’;
  not     = ’not’;
  set     = ’set’;
  call    = ’call’;
  is      = ’is’;
  has     = ’has’;
  element = ’element’;
  isempty = ’isEmpty’;
  size    = ’size’;
  hint    = ’hint’;

  // Basic Types
  bool_type   = ’Boolean’;
  int_type    = ’Int’;
  string_type = ’String’;

  // Literals
  bool_constant   = ’true’ | ’false’;
  int_constant    = digit+;
```

```
  string_constant = '"' [not_cr_lf - '"']* '"';

  // Identifiers
  identifier  = al alnum*;

  // Whitespace & comments
  blank  = (' ' | eol | tab)+;
  comment = onelined_comment | multiline_comment;


Ignored Tokens
  blank, comment;


Productions
  odl_start = {proposition} proposition |
              {named_predicate} named_predicate_declaration;

  named_predicate_declaration = identifier l_par type_list r_par
                                assign ccl_proposition;

  proposition =
    {unop}    unary_proposition |
    {and}     proposition and unary_proposition |
    {or}      proposition or unary_proposition |
    {implies} proposition implies unary_proposition |
    {equiv}   proposition equiv unary_proposition;

  // In a ccl proposition no 'post_relation' (result ...) and
  // no quantors 'context' and 'new' are allowed
  ccl_proposition =
    {unop}    ccl_unary_proposition |
    {and}     ccl_proposition and ccl_unary_proposition |
    {or}      ccl_proposition or ccl_unary_proposition |
    {implies} ccl_proposition implies ccl_unary_proposition |
    {equiv}   ccl_proposition equiv ccl_unary_proposition;

  unary_proposition =
    {neg}             neg unary_proposition |
    {quantifier}      quantifier variable_definition dot
                      unary_proposition |
    {new_quantifier}  new_quantifier model_element_variable_definition
                      dot unary_proposition |
    {named_predicate} named_predicate_call |
    {term}            term;

  ccl_unary_proposition =
    {neg}               neg ccl_unary_proposition |
    {quantifier}        ccl_quantifier variable_definition dot
                        ccl_unary_proposition |
    {named_predicate}   named_predicate_call |
```

```
    {term}              ccl_term;

  quantifier =
    {ccl}     ccl_quantifier |
    {context} context context_extension?;

  ccl_quantifier =
    {forall}  forall |
    {exists}  exists;

  new_quantifier = new;

  context_extension = l_bracket hint_extension? r_bracket;

  hint_extension = hint equal string_constant_expr_list;

  string_constant_expr_list = string_constant_expr
                              string_constant_expr_list_tail*;

  string_constant_expr_list_tail = comma string_constant_expr;

  term =
    {basic}   basic_proposition |
    {par}     l_par proposition r_par;

  ccl_term =
    {basic}   ccl_basic_proposition |
    {par}     l_par ccl_proposition r_par;

  basic_proposition =
    {relation} relation |
    {bool}     bool_proposition;

  ccl_basic_proposition =
    {relation}   ccl_relation |
    {bool}       bool_proposition;

  bool_proposition =
    {equal}          equal_expression |
    {bigger_smaller} bigger_smaller_expression |
    {is_empty}       isempty l_par expression r_par |
    {constant}       bool_constant_expr;

  // Comparisons
  bigger_smaller_expression = [l_expr]:expression comparison_operator
                              [r_expr]:expression;

  comparison_operator =
    {bigger} bigger |
    {smaller} smaller |
    {bigger_or_equal} bigger_or_equal |
```

```
    {smaller_or_equal} smaller_or_equal;

  // Relations
  relation =
    {is}  pre_relation |
    {mod} post_relation;

  ccl_relation =
    {is}  pre_relation;

  pre_relation = is call_expression;

  post_relation =
    {add} result has call_expression |
    {del} result not has call_expression;

  // Call expressions
  call_expression = identifier l_par args r_par;

  named_predicate_call = call call_expression;

  args = arglist?;

  arglist = arg arglist_tail*;

  arglist_tail = comma arg;

  arg = expression;

  // Expressions
  non_constant_expression =
    {fct}         functional_expression |
    {sel}         selector_expression |
    {var}         defined_variable;

  expression =
    {non_constant} non_constant_expression |
    {arithmetic}   arithmetic_expression |
    {constant}     constant_expression;

  constant_expression =
    {bool}   bool_constant_expr |
    {string} string_constant_expr;

  bool_constant_expr = bool_constant;

  int_constant_expr = sign? int_constant;

  string_constant_expr = string_constant;

  sign =
```

```
    {unary_plus}  plus |
    {unary_minus} minus;

  functional_expression = call_expression;

  selector_expression = defined_variable dot selection;

  selection = selector selection_tail*;

  selection_tail = dot selector;

  selector = identifier;

  defined_variable = variable;

  variable = identifier;

  equal_expression = [l_expr]:expression equal [r_expr]:expression;

  // Arithmetic expressions
  arithmetic_expression =
    {factor} factor |
    {plus}   arithmetic_expression plus factor |
    {minus}  arithmetic_expression minus factor;

  factor =
    {arithmetic_term} arithmetic_term |
    {mult}            factor mult arithmetic_term;

  arithmetic_term =
    {constant}   int_constant_expr |
    {expression} l_par arithmetic_expression r_par |
    {set_size}   size l_par expression r_par;

  // Variable definitions
  variable_definition = variable colon type;

  model_element_variable_definition = variable colon model_element_type;

  // Type definitions
  type = {product}    product_type |
         {restricted} restricted_type_definition |
         {set}        set_type_definition |
         {set_variable}   element defined_variable |
         {set_expression} element l_par expression r_par;

  restricted_type_definition = l_brace variable colon type v_line
                               ccl_proposition r_brace;

  set_type_definition = set type;
```

```
product_type = {unary} unary_type |
               {list}  l_par type_list r_par;

type_list = identifier colon type type_list_tail*;

type_list_tail = comma identifier colon type;

unary_type = {basic} basic_type |
             {model} model_element_type;

basic_type = {boolean} bool_type |
             {integer} int_type |
             {string}  string_type;

model_element_type = identifier;
```

Literaturverzeichnis

[BLS00] P. Braun, H. Lötzbeyer, O. Slotosch. *Developing Embedded Systems with AutoFocus/Quest*. Institut für Informatik, Technische Universität München, 2000.

[BLS01] P. Braun, H. Lötzbeyer, O. Slotosch. *Project QUEST. Integrated Metamodels for AutoFocus*. Institut für Informatik, Technische Universität München, 2001.

[QuestUser] P. Braun, H. Lötzbeyer, O. Slotosch. *Quest Users Guide, Version 1.0*, 2003.

[QuestDev] P. Braun, H. Lötzbeyer, O. Slotosch. *QUEST Developers Guide*, 2003.

[TACAS02] P. Braun, F. Huber, B. Schätz, A. Wisspeintner. *Preserving Model Consistency in Software Development*. Institut für Informatik, Technische Universität München, 2002.

[TACAS00] P. Braun, H. Lötzbeyer, B. Schätz, O. Slotosch. *Consistent Integration of Formal Methods*. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000), 2000.

[SableCC] É. Gagnon. *SableCC, An Object-Oriented Compiler Framework*. School of Computer Science, McGill University, Montreal, 1998.

[GammaEtAl] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996.

[HS02] F. Huber, B. Schätz. *Integrated Development of Embedded Systems with AutoFocus*. Institut für Informatik, Technische Universität München, 2002.

[Kemper] A. Kemper, A. Eickler. *Datenbanksysteme. Eine Einführung*. R.Oldenbourg Verlag München Wien, 1999, 3. Auflage.

[OCL] OMG. *Unified Modeling Language: OCL version 2.0*. ptc/03-08-08, Object Management Group (OMG), http://www.omg.org, 2003.

[Pasch] D. Pasch. *Konzeption und Implementierung eines ODL-Interpreters für das AutoFocus/Quest CASE-Werkzeug*. Bachelor Thesis, Technische Universität München, 2002.

[Sch02] B. Schätz. *Process Support within the AutoFocus/Quest Application Framework AQuA – Internal Draft Version*. Institut für Informatik, Technische Universität München, 2002.

[Sch01] B. Schätz. *The ODL Operation Definition Language and the AutoFocus/Quest Application Frameword AQuA. Technischer Bericht TUM-I0111*. Institut für Informatik, Technische Universität München, 2001.

[SBHW03] B. Schätz, P. Braun, F. Huber, A. Wisspeintner. *Checking and Transforming Models: Methodological Aspects of Model-Based Development*. Institut für Informatik, Technische Universität München, 2003.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, J. Philips. *Model-based Development of Embedded Systems. Technical Report TUM-I0402*. Institut für Informatik, Technische Universität München, 2002.

[Validator] *Validator Manual. Version 1.4.2.b*. Validas Model Validation AG, 2003.

[JavaAPI] Java™ *2 Platform, Standard Edition, v 1.4.2 API Specifications*. Sun Microsystems Inc., 2003.

[ODLAPI] *ODL API Specifications v 1.1*. Generated by javadoc from sources, 2003.

[AFHome] *Die AutoFocus Homepage*, http://autofocus.in.tum.de. Fakultät für Informatik, Technische Universität München.

[QUESTHome] *Software Development Project: QUEST*, http://www4.in.tum.de/proj/quest. Fakultät für Informatik, Technische Universität München.

[Tracht] D. Trachtenherz. *Konzeptmodellbasierte Manipulation von Systemspezifikationen*. Systementwicklungsprojekt, Technische Universität München, 2002.